Technology
Arts Sciences
TH Köln

Automatische Bewertung von Codequalität innerhalb eines Code Reviews

Masterarbeit

ausgearbeitet von

Sascha Lemke

zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

vorgelegt an der

Technischen Hochschule Köln
Campus Gummersbach
Fakultät für Informatik und
Ingenieurwissenschaften

im Studiengang
Medieninformatik

Erster Prüfer: Prof. Dr. Christian Kohls
Technische Hochschule Köln

Zweiter Prüfer: Prof. Dr. Kristian Fischer
Technische Hochschule Köln

Gummersbach, im November 2020

Inhaltsverzeichnis

Kurzfassung

Eine gängige Form der Qualitätskontrolle von Quellcode sind Code Reviews. Der Fokus von Code Reviews liegt allerdings oft auf syntaktischer Analyse, wodurch weniger Zeit für eine semantische Überprüfung bleibt und zusätzliche Kosten verursacht werden. Code Reviews lassen sich zwar teilweise durch „Linter“ automatisieren, dennoch können sie nur syntaktische Fehlermuster identifizieren, welche vorher definiert wurden. Zudem kann ein Linter nur darauf hinweisen, dass möglicherweise ein Fehler vorliegt, da die Fehler nicht durch logische Inferenz ermittelt werden.

Die vorliegende Arbeit prüft, ob ein Deep Learning Modell den regelbasierten Ansatz von Lintern ablösen und die semantische Ebene erschließen kann. Dazu wurde eine Stichprobe von Java Methoden zusammengestellt und im Anschluss mit einem Supervised Learning Ansatz binär klassifiziert. Da die Analyse von Quellcode der Textanalyse stark ähnelt wird ein gängiger Ansatz für Textklassifikation verwendet. Dadurch kann gezeigt werden, dass eine Präzision von 85% bei der Erkennung von Quellcodeproblemen durch Deep Learning möglich ist.

Abstract

A common form of quality control of source code are code reviews. However, code reviews often focus on syntactic analysis, leaving little time for semantic review and increase costs. Although code reviews can be partially automated by a programm called „linter“, they can only identify syntactic error patterns that have been previously defined. In addition, a linter can only indicate that there may be an error, because the errors are not determined by logical inference.

This work examines whether a deep learning model can replace the rule-based approach of linters and open up the semantic level. For this purpose, a sample of Java methods was compiled and then binary classified with a supervised learning approach. Since the procedure of the source code analysis closely resembles the procedure of text analysis, a common approach to text classification is used. This demonstrates that 85% accuracy is possible in detecting source code problems through deep learning.

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das Finden von Fehlern in Quellcode und die Verbesserung dessen Qualität ist auch heute noch ein wichtiges Thema in der Informatik. Über die letzten Jahrzehnte haben unzählige Verbesserungen in Entwicklungsprozessen, -methoden und -werkzeugen dafür gesorgt, dass viele häufig auftretende Fehler besser identifiziert oder sogar komplett vermieden werden können. Eine Methode zum Finden von Fehlern ist das Code Review, welches mittlerweile als Industriestandard gilt und in großen Unternehmen zum Alltag gehört. Bei einem Code Review wird der Quellcode eines Entwicklers durch andere Entwickler auf Fehler überprüft. Neben der Fehlersuche existieren bei einem Code Review weitere Ziele, wie die Einhaltung von Standards, Wissensübertragung zwischen Entwicklern oder die Diskussion alternativer Lösungsansätze für das vorliegende Problem. Allerdings stellen Code Reviews hohe Anforderungen an den Entwickler, welcher den Quellcode prüft. So muss dieser potentielle Fehlerquellen, Qualitätsstandards und alternative Lösungsansätze kennen, um den Quellcode bewerten zu können. Hinzu kommt, dass eine menschliche Bewertung nicht nur fehleranfällig und subjektiv ist, sondern auch Zeit und Geld kostet.

1.1 Motivation

Mit Hilfe von Lintern lassen sich durch statische Analyse viele syntaktisch nachweisbare Probleme von Quellcode erkennen. Ein Linter untersucht den Quellcode auf typische Fehlermuster und informiert den Entwickler durch entsprechende Fehlermeldungen. Bei der Untersuchung wird das Programm allerdings nicht ausgeführt, sondern eine abstrakte Form des Quellcodes mit Hilfe von vorher definierten Regeln überprüft. Auch wenn dadurch gewisse Fehler gefunden oder Standards eingehalten werden können, so hat dieser Ansatz einige Schwächen. Da das Programm nicht ausgeführt wird, werden Fehler, welche nur zur Laufzeit auftauchen, nicht erkannt. Außerdem kann die statische Analyse anhand der Syntax des Quellcodes nicht alle Probleme identifizieren, da Probleme von Quellcode auch auf semantischer Ebene existieren. Daher ist eine Forderung nach besseren Werkzeugen zur Fehlererkennung auch heute noch relevant.

Ein neues Themenfeld, welches Verbesserung in diesem Bereich verspricht, ist die Anwendung von Machine Learning auf Quellcode. In den letzten Jahren haben zahlreiche Machine Learning Modelle diverse State of the Art Ergebnisse in ver-

schiedenen Disziplinen erreicht. Eine Disziplin ist das Natural Language Processing (NLP), welche sich mit der Interpretation, Verarbeitung und Generation von natürlicher Sprache auseinandersetzt. Ursprünglich war die Untersuchung von Text durch die Formulierung von Regeln geprägt, welche natürliche Sprache beschreiben sollten. Im Laufe der Zeit zeigte sich allerdings, dass ein statistischer Ansatz deutlich bessere Ergebnisse lieferte. Dieser Trend ebnete im späteren Verlauf den Weg für Machine Learning, dem heutigen dominanten Ansatz bei der Analyse von natürlicher Sprache. Dabei erwies sich Deep Learning als ein besonders erfolgreicher und universeller Ansatz, da Deep Learning in der Lage ist die relevanten Features selbstständig zu extrahieren. Durch diesen Ansatz konnten weitreichende Verbesserungen bei der Sprachmodellierung, Textklassifikation und Textgeneration erreicht werden. Die Ähnlichkeiten zwischen der Auswertung von Quellcode mit statischer Analyse und dem Natural Language Processing führten zu der Hypothese, dass Quellcode ähnliche Merkmale wie Text aufweisen und sich der Erfolg übertragen lassen könnte. Auf Basis dessen baute sich in den letzten Jahren ein neues Forschungsfeld auf, welches stellenweise als „Big Code“ bezeichnet wird. Dieses Forschungsfeld versucht die Natürlichkeit von Quellcode zu nutzen, um probabilistische Modelle zu entwickeln und diese für einen bessere Entwicklungsprozess zu nutzen. Deep Learning ist bei natürlicher Sprache in der Lage, semantische Informationen durch statistische Merkmale zu entdecken. Sollte sich dies auf Quellcode übertragen lassen, dann wäre es möglich die semantische Ebene für Linter zu erschließen.

1.2 Zielsetzung

Der erste Schritt bei Code Reviews und Lintern ist die Identifizierung eines Problems anhand von Codequalitätsmerkmalen. Die vorliegende Arbeit versucht durch die Verwendung von Deep Learning das Potential für eine automatische Erkennung von semantischen Codequalitätsmerkmalen zu ergründen. Dies könnte Lintern ermöglichen semantische Probleme im Quellcode zu erkennen, wodurch Code Reviews sich weiter automatisieren lassen könnten. Um dies zu überprüfen, wird eine Stichprobe manuell klassifiziert, welche gute und schlechte Java Methoden enthält. Auf Basis dieser Stichprobe wird ein Deep Learning Modell entwickelt, welches neue Java Methoden automatisch als „gut“ oder „schlecht“ klassifiziert. Im Anschluss werden die Ergebnisse der automatischen Klassifikation quantifiziert, um den Erfolg eines solchen Ansatzes zu messen. Zusätzlich wird eine Stichprobe mit Java Methodennamen zusammengestellt, um zu überprüfen, ob Namen isoliert klassifiziert werden können, da Namen ein besonders relevantes Merkmale für Codequalität sind. Da Deep Learning die relevanten Merkmale selbstständig extrahiert, ist für diese Arbeit keine manuelle Modellierung der Codequalität notwendig. Dennoch ist ein ausreichendes Verständnis der Merkmale und der Daten wichtig, um spätere Ergebnisse richtig zu interpretieren. Daher versucht diese Arbeit neben der Klassifikation einen statistischen Einblick in den Datensatz zu erhalten und zu ermitteln, welche Faktoren die Auswahl der Merkmale beeinflusst haben könnten.

1.3 Aufbau

Nachdem die Motivation und Zielsetzung der Arbeit in diesem Kapitel beschrieben wurden teilt sich die Arbeit in sechs weitere Kapitel auf.

In Kapitel 2 werden Code Reviews genauer betrachtet und der aktuelle Stand der Technik bezüglich der Analyse von Quellcode dokumentiert. Daraufhin werden dessen Limitierungen aufgezeigt und die Theorie „Naturalness of Source Code“ eingeführt, um die Motivation für Machine Learning Modelle auf Basis von Quellcode zu verdeutlichen.

Kapitel 3 beschreibt, wie sich Codequalität in objektorientierten Sprachen erkennen lässt und welche Merkmale diese begünstigen. Dadurch wird der theoretische Rahmen für die manuelle Klassifikation der Java Methoden geschaffen und die Kriterien für die Untersuchung in Kapitel 5 definiert.

Kapitel 4 erläutert die theoretischen Grundlagen binärer Textklassifikation durch Deep Learning. Dadurch wird die Basis für das methodische Vorgehen geschaffen. Zum Abschluss wird der aktuelle Forschungsstand dokumentiert.

In Kapitel 5 wird die Methodik der Arbeit dokumentiert. Zu Beginn werden Forschungsfragen definiert, welche die Zielsetzung aus Kapitel 1 reflektieren. Im Anschluss wird beschrieben, wie die Stichprobe halbautomatisch erhoben und mit Deep Learning ausgewertet wurde.

Im Kapitel 6 wird das Ergebnis der Untersuchung aus Kapitel 5 ausführlich vorgestellt und dokumentiert. Das Ergebnis wird außerdem diskutiert und interpretiert.

Im siebten und letzten Kapitel der Arbeit werden die Forschungsfragen anhand der Ergebnisse aus Kapitel 6 im Rahmen eines Fazits beantwortet. An dieser Stelle werden auch Limitationen der Arbeit aufgezeigt und ein Ausblick über weitere Möglichkeiten zur Forschung gegeben.

2 Analyse von Quellcode

Um Code Reviews verbessern zu können ist es notwendig den Prozess zu verstehen und den aktuellen Stand automatischer Quellcodeanalyse zu kennen. Dieses Kapitel liefert die relevanten theoretischen Grundlagen und verdeutlicht die Motivation eines Deep Learning Ansatzes.

2.1 Code Reviews

Bei Code Reviews handelt es sich um eine anerkannte Methode zum Finden von Fehlern. Sie basiert auf einer informellen und regelmäßigen Überprüfung von Quellcode [7]. Zusätzlich sollen Code Reviews dabei helfen die Qualität des Quellcodes zu erhöhen, Wissen zu teilen oder Konventionen einzuhalten [7]. Die Überprüfung wird dabei von Entwicklern durchgeführt, welche den Quellcode nicht selbst geschrieben haben und kann manuell oder softwareunterstützt statt finden [8]. Dabei kann es sich um Entwickler aus dem eigenen Team, einem anderen Team oder sogar um extern beschäftigte Entwickler handeln. In großen Softwareunternehmen gehören Code Reviews zum Alltag eines Entwicklers [7].

Moderne Code Reviews sind aus den Code Inspektionen der 70er Jahre hervorgegangen, welche sehr formell und stark strukturiert waren [7]. So formulierte Fagan einen Prozess auf Basis von Gruppenmeetings, bei dem der Quellcode Zeile für Zeile auf eine ganze Reihe von Kriterien überprüft wurde, welche in Checklisten festgehalten wurden [7] [21]. Durch dieses Vorgehen sollten Fehler im Quellcode möglichst früh im Entwicklungsprozess entdeckt werden, da sich zeigte, dass spät entdeckte Fehler hohe Kosten verursachten [21]. Allerdings stellte sich die Durchführung als sehr aufwendig heraus, weswegen im Laufe der Zeit viele formelle Anforderungen fallen gelassen wurden. Die Distanzierung führt außerdem dazu, dass Code Reviews immer mehr zum kollaborativen Lösen von Problemen eingesetzt werden [7].

Dies führte zu einem flexibleren Prozess [7] [9], welcher in Abbildung 2.1 abgebildet ist. Die Ausgangsbasis des Prozesses ist der zu untersuchende Quellcode. Dabei kann es sich um ein komplettes Programm handeln oder um eine Reihe von Änderungen innerhalb eines Versionierungssystems. Im Anschluss wird der Quellcode überprüft und es werden mögliche Fehler, Probleme oder Verstöße gegen Konventionen festgehalten. Um die Überprüfung möglichst unvoreingenommen durchzuführen, sollte der prüfende Entwickler nicht an der Entwicklung beteiligt gewesen sein. Danach hat der Autor die Möglichkeit seinen Quellcode zu verbessern. Dadurch kann er

auch aus den Fehlern lernen, um nicht ähnliche Fehler in folgenden Reviews zu wiederholen [21]. Die Prüfung wird dabei solange fortgesetzt, bis alle Kriterien erfüllt sind. Das Ergebnis dieses iterativen Prozesses ist Quellcode, welcher durch das restliche Team akzeptiert wird und einem gemeinsam festgelegten Qualitätsstandard folgt. Allerdings kann der Qualitätsstandard und die Durchführung in der Praxis aufgrund der Flexibilität des Prozesses und der Kriterien stark variieren.

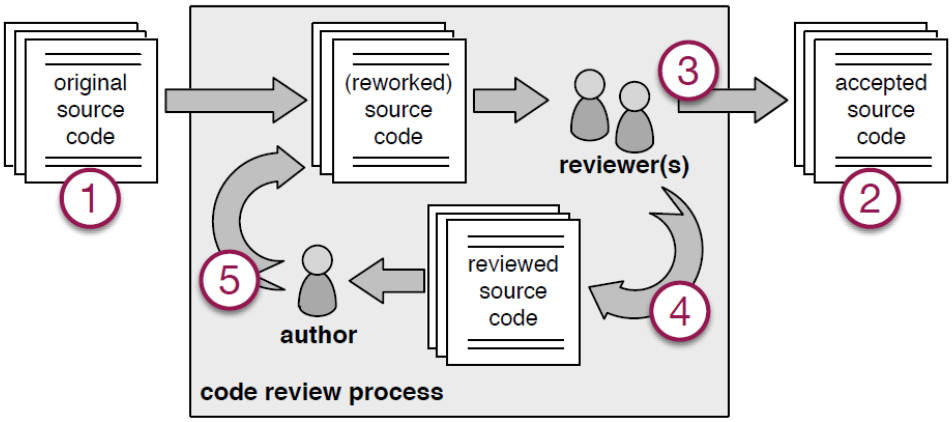

Abbildung 2.1: Ablauf eines Code Review aus [7]

Moderne Code Reviews sehen sich allerdings einigen Herausforderungen gegenüber. Die größte Herausforderung ist der notwendige zeitliche Aufwand, um den Quellcode zu überprüfen [7]. Außerdem müssen die Prüfer über das notwendige Wissen verfügen, um Probleme im Quellcode entdecken zu können [7]. Dies führt zu steigenden Kosten, da die Weiterbildung der Entwickler und der zeitliche Aufwand entsprechend finanziert werden müssen. Zusätzlich zeigte eine Studie, dass ungefähr 75% der gefundenen Probleme in Code Reviews das Verhalten der Software nicht beeinflussen [41]. Auch Entwickler bei Microsoft sind der Ansicht, dass Code Reviews den Fokus oft zu sehr auf kleinere Logikfehler lenken und dadurch kaum Platz für die Diskussion tieferer Designprobleme vorhanden ist [7].

Einige Effekte dieser Herausforderungen lassen sich allerdings durch Automatisierung mindern. Im Rahmen von Code Reviews wird oft die statische Analyse [27] eingesetzt, um häufige Fehler zu erkennen oder Konventionen durchzusetzen. Bei der statischen Analyse wird der Quellcode untersucht, ohne diesen auszuführen [27]. Die Analyse kann entweder lokal durchgeführt werden oder Teil eines agilen Prozesses sein [8]. Dabei kann eine statische Analyse allerdings nie ein Code Review ersetzen, sondern nur als Voruntersuchung dienen, um die Überprüfung durch den Entwickler zu verbessern [27].

2.2 Statische Analyse

Die statische Analyse bezeichnet eine Reihe von automatisierten Methoden, um Quellcode zu analysieren ohne diesen auszuführen [20]. Die Analyse von Quellcode ist ein Untersuchungsgegenstand der Informatik seit Quellcode existiert, weswegen viele der Methoden auch in Compilern zu finden sind [28]. Neben der statischen Analyse existiert auch die dynamische Analyse, welche das Programm ausführt und dessen Verhalten beobachtet, um Fehler zu finden [51].

2.2.1 Linter

Das erste statische Analyseprogramme war das Programm „Lint“ [33], welches C Programme auf Fehler untersuchte. Aus diesem Grund werden statische Analyseprogramme oft auch als „Linter“ bezeichnet. Linter arbeiten auf der Premisse, dass Menschen die gleichen Fehler wiederholen und deswegen Fehlermuster im Quellcode zu finden sind [40]. Aus diesem Grund komplementieren Linter auch Code Reviews und Tests, da bereits vorab mögliche Probleme oder Defekte automatisch und früh im Entwicklungsprozess aufgedeckt werden können [8] [51].

Die meisten Linter arbeiten auf Basis des Quellcodes, allerdings existieren auch Linter die auf Bytecode arbeiten [13] [40]. Auch wenn die Implementierungsdetails von Lintern variieren können, so arbeiten diese üblicherweise sehr ähnlich [40]. Zu Beginn wird der Quellcode eingelesen, um eine abstraktere Datenstruktur zu generieren, mit welcher sich im Anschluss Muster erkennen lassen [40]. Die Fehlermuster werden durch vorher definierte Regeln erkannt, weswegen die Qualität eines Linters auch von diesen abhängt [27].

Um Fehler zu erkennen wird allerdings eine geeignete Datenstruktur benötigt, welche die Analyse des Quellcodes und das Speichern von zusätzlichen Daten erlaubt. Eine Datenstruktur, welche diesen Anforderungen gerecht wird ist der Abstract Syntax Tree.

2.2.2 Abstract Syntax Tree

Ein Abstract Syntax Tree (AST) [28] ist eine Baumstruktur, welche die syntaktische Struktur des Quellcodes repräsentiert. Abbildung 2.2 zeigt ein Beispiel für einen AST. Jeder Knoten des Baumes beschreibt dabei ein Konstrukt des Quellcodes. Durch diese Transformation können die vielen Vorteile von Baumstrukturen, wie etwa Traversierung, für die statische Analyse verwendet werden. Außerdem können zusätzliche Informationen für Analyse- oder Optimierungszwecke annotiert werden [28]. Ein Abstract Syntax Tree kann durch die syntaktische Analyse des Quellcodes generiert werden, welche auch als Parsing bezeichnet wird [28]. Abbildung 2.3 zeigt, wie sich ein AST generieren lässt.

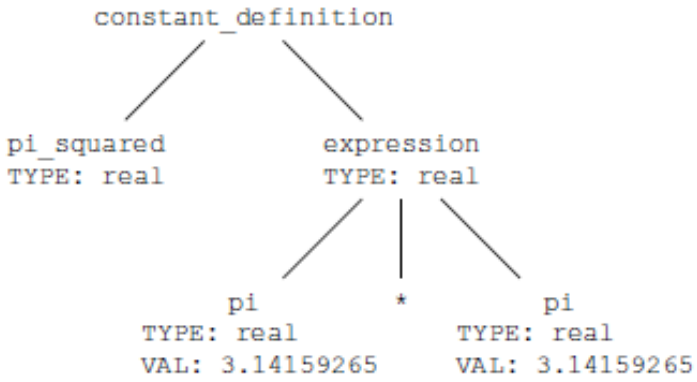

Abbildung 2.2: Abbildung eines annotierten Abstract Syntax Tree aus [28]

Zu Beginn wird der Quellcode einer lexikalischen Analyse unterzogen. Dabei werden sogenannte Token gewonnen, welche einzelne Bestandteile des Quellcodes repräsentieren [28]. Die Konstruktion der Token folgt nicht unbedingt allgemeingültigen Regeln [1]. Ein Ansatz zur Konstruktion von Token ist: wenn eine Zeichensequenz links und rechts von Leerzeichen getrennt wird, dann ist es ein Token [28]. Allerdings können beispielsweise Strings auch Leerzeichen enthalten, welche aber dennoch als ein ganzes Token angesehen werden [28]. Die Aufteilung des Quellcodes in einzelne Token erfolgt üblicherweise über Reguläre Ausdrücke, welche die Muster der Token einer Programmiersprache beschreiben [28]. Ein Token besteht üblicherweise aus einer Nummer (Klasse), einem String (Quellcode) und Positionsinformationen [28].

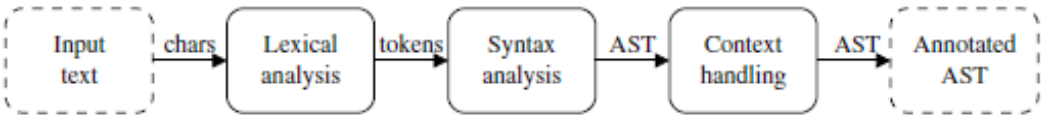

Abbildung 2.3: Generation einen Abstract Syntax Trees aus [28]

Die nun gewonnen Token werden im Anschluss einer syntaktischen Analyse unterzogen. Bei dieser Analyse werden die Token zu einem Baum zusammengefasst [28]. Das Finden von Strukturen in Token wird auch als Parsing bezeichnet und von einem sogenannten Parser durchgeführt [28]. Dabei gibt es zwei verschiedene Ansätze die Parser verfolgen können: top-down oder bottom-up [28]. Diese Ansätze unterscheiden sich in der Reihenfolge, wie sie die Knoten des Abstract Syntax Tree konstruieren [28]. Beim top-down Ansatz werden die Knoten in pre-order konstruiert während der bottom-up Ansatz die Knoten in post-order konstruiert [28]. Top-down Parser können entweder von Hand geschrieben oder generiert werden, bottom-up Parser hingegen nur generiert werden [28]. Die automatische Generierung eines Parser erfolgt durch kontextfreie Grammatiken, welche die Programmiersprache beschreiben [28]. Dies wurde erstmals von Lewis und Steams beschrieben [38].

Der nun vorhandene Abstract Syntax Tree verfügt allerdings nur über kontextfreie Informationen [28]. Kontextinformationen, wie das Prüfen von Parametern einer Methode gegenüber der Methodendeklaration, gehören nicht dazu [28]. Diese werden in einem letzten Schritt überprüft und annotiert, was als Context Handling bezeichnet wird [28]. Die annotierten Informationen werden auch als Attribute eines Knotens bezeichnet [28]. Die Informationen eines Attributs werden durch ein Name-Wert Paar dargestellt [28]. Context Handling kann automatisch von einer Attributgrammatik durchgeführt oder durch einen Entwickler definiert werden [28]. Eine Attributgrammatik ist eine Erweiterung einer kontextfreien Grammatik, welche die notwendigen Regeln für das Context Handling enthält [28]. Attributgrammatiken wurden von Donald Knuth [36] eingeführt und erweitern kontextfreie Grammatiken um zwei Konzepte: Attribute und Evaluierungsregeln [28]. Mit den Attributen können semantische Informationen definiert werden, welche in Abhängigkeit der Evaluierungsregeln annotiert werden [36]. Dabei unterscheidet man zwischen synthetisierten und geerbten Attributen [36]. Die synthetisierten Attribute basieren auf den Attributen der Nachkommen [36]. Geerbte Attribute hingegen basieren auf den Attributen der vorhergehenden Symbole [36].

2.2.3 Limitierungen

Der nun vorhandene annotierte Abstract Syntax Tree kann für die Erkennung von Fehlern im Quellcode verwendet werden. Dadurch können Linter verschiedene Probleme finden wie z.B. illegale Operationen oder ungenutzen Quellcode [20]. Doch auch wenn ein Linter Fehler im Quellcode finden kann, garantiert dies nicht, dass ein Programm korrekt funktioniert, sondern nur, dass es nicht aufgrund von unerwarteten Problemen zur Laufzeit abbricht [20]. Des Weiteren garantieren Linter durch das Entfernen von Fehlern keine hohe Programmqualität [40]. Ein Linter kann außerdem nie alle Fehler finden, da er auch nur Fehler findet, welche er bereits kennt [27]. Allerdings hat eine automatische Untersuchung den Vorteil, dass sie gegenüber einer menschlichen objektiver ist, da Menschen durch den Zweck des Quellcodes beeinflusst werden [32].

Darüber hinaus sind viele Fehler nicht entscheidbar, wodurch ein Linter nur berichten kann, dass möglicherweise ein Problem vorliegt [13] [20]. So kann es zu falschpositiven Ergebnissen kommen, welche ein Problem statischer Analyse sind [27]. Ein Entwickler muss dann entscheiden, ob es sich wirklich um ein Problem handelt [27]. Hinzu kommt, dass viele Probleme sich gar nicht durch Regeln beschreiben lassen, da sie ihren Ursprung in den Anforderungen des Programms haben [27]. Deswegen erfordern die Ergebnisse einer statischen Analyse immer die Beurteilung durch einen Entwickler [27].

2.3 Naturalness of Source Code

Zusammenfassend lassen sich drei große Probleme statischer Analyse formulieren. Als erstes sorgt die Definition von Regeln dafür, dass der Ansatz fragil ist und von der Qualität der Regeln abhängt. Als zweites müssen Merkmale von Fehlern herausgearbeitet und in geeignete Regeln transformiert werden. Drittens werden nicht alle Probleme erkannt, da viele einen semantischen Ursprung haben.

Die Vorgehensweise der statischen Analyse zeigt allerdings deutliche Parallelen zu früheren Ansätzen des Natural Language Processing (NLP). NLP ist eine Querschnittsdisziplin, welche sich aus den Forschungsbereichen Computerlinguistik und künstlicher Intelligenz zusammensetzt und beschäftigt sich mit der Analyse von natürlichen Sprachen [30]. Viele der in Abschnitt 2.2 beschriebenen Methoden finden sich auch im NLP wieder. Allerdings wechselte NLP in den 80er und 90er Jahren von Regeln zu statistischen Daten natürlicher Sprache [30]. Dies lag an der großen Vielfalt und Dynamik natürlicher Sprache, wodurch nie ausreichende und qualitative Regeln formuliert werden konnten [48]. Der statistische Ansatz liefert deswegen bessere Ergebnisse, weil dessen Qualität durch eine steigende Menge repräsentativer Daten zunimmt [48]. Außerdem kann dabei mit unbekannten Daten besser umgegangen werden [48]. Dieser Wechsel sorgte im späteren Verlauf für die Adaption vieler Machine Learning Verfahren, welche zum dominanten Ansatz des NLP wurden [17].

Dieser Verlauf könnte sich auch auf die Analyse von Quellcode übertragen lassen, denn Programmiersprachen dienen nicht nur zur Kommunikation zwischen Mensch und Maschine, sondern auch für die Kommunikation von Entwicklern untereinander [3]. Dieser Gedanke führte zur Formulierung der „Naturalness of Source Code“ Theorie von Allamanis et al. [3]:

> Programming languages, in theory, are complex, flexible and powerful, but, „natural“ programs, the ones that real people actually write, are mostly simple and rather repetitive; thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks.

So konnte gezeigte werden, dass Quellcode ähnliche Eigenschaften wie natürliche Sprache aufweist [3]. Des Weiteren existiert die Hypothese, dass Quellcode aufgrund der Natürlichkeit tiefere semantische Informationen manifestiert und diese abgeleitet werden können [29]. Dies führt zu einem neuen Forschungsthema bei dem Machine Learning angewandt wird, um probabilistische Modelle zu schaffen, welche den Entwicklungsprozess verbessern könnten [3]. Dieses Forschungsthema wird stellenweise auch als „Big Code“ bezeichnet [3] [11].

3 Codequalität

Um die Qualität von Quellcode beurteilen zu können, ist es eine notwendige Voraussetzung guten von schlechtem Quellcode unterscheiden zu können. Aus diesem Grund wurde eine Literaturrecherche durchgeführt, um die Qualität von Quellcode zu beschreiben. Die hier beschriebene Codequalität beschäftigt sich mit Merkmalen guten objektorientierten Quellcodes und dient zur Auswertung des Quellcodes in Kapitel 5 und 6. Allerdings werden nur die Inhalte dokumentiert, welche einen Einfluss auf den Entwurf einer Methode haben. Codequalität ist ein sehr großes Thema und dieses Kapitel liefert die Grundlage für die binäre Klassifikation.

Die Literaturrecherche basiert auf Veröffentlichungen von bekannten Softwareentwicklern, welche ähnliche Gedanken zu gutem Quellcode teilen [43]. Basierend auf diesen Gedanke veröffentliche Martin et al. ein Buch namens „Clean Code“, welches als Handbuch für guten objektorientierten Quellcode gilt. Abschnitt 3.1 stellt einen kurzen Überblick der relevanten Clean Code Inhalte vor. Im Anschluss werden „Code Smells“ von Fowler et al. vorgestellt, welche Muster beschrieben, die auf ein tieferes Problem im Quellcode hinweisen. Daraufhin wird ein kurzer Überblick über relevante Architekturentscheidungen und Designprinzipien von Quellcode gegeben, welche einen Einfluss auf Klassen und Methodendesign haben.

3.1 Clean Code

Bei Clean Code handelt es sich nicht um eine klare Definition oder um ein allumfassendes Regelwerk, welchem es ausnahmslos zu folgen gilt. Clean Code ist eine Sammlung von Richtlinien für objektorientierte Programmiersprachen, welche guten Quellcode hervorbringen sollen. Diese Sammlung basiert auf dem Buch „Clean Code“ von Martin et al., welche anhand vieler Beispiele deutlich macht, wie sich guter Quellcode umsetzen lässt [43]. Die folgende Darstellung von Clean Code basiert komplett auf Martin et al.

3.1.1 Sinnvolle Namen

Ein erster und wichtiger Punkt im Rahmen von Clean Code ist die sinnvolle Benennung von Variablen, Methoden und anderen Elementen einer Programmiersprache. Dabei ist es wichtig Namen zu verwenden, welche den Sinn und die Absicht einer Methode oder Klasse klar beschreiben.

Um sinnvolle und eindeutige Namen zu verwenden existieren einige Richtlinien, welche der Vermeidung von Problemen dienen. Die erste Richtlinie ist die Vermeidung von falschen Informationen durch schlechte Namen. Falsche Informationen können durch die Verwendung von Abkürzungen oder falschen Konzeptbezeichnungen auftreten. Abkürzungen sind problematisch, weil sie je nach Kontext und Verständnis der Entwickler variieren können und dadurch nicht gegeben ist, dass jeder Entwickler das gleiche Konzept darunter versteht. Auch die Verwendung von unscharfen Bezeichnungen, wie `users` gegenüber `userList`, kann zu Problemen führen. Idealerweise würde man unter `users` eine Menge von Nutzerobjekten verstehen, welche sich beispielsweise in einem Array oder einer Liste befinden können. Der Name `userList` macht in diesem Fall deutlich, das es sich um eine Liste von Nutzern handelt und dementsprechend eine Liste als Datenstruktur verwendet werden sollte.

Besonders problematisch wird es, wenn der Entwickler nicht nur unscharfe Bezeichnungen, sondern auch Konzepte aus fremden Domänen verwendet oder bereits verwendete Konzepte aus der aktuellen Domäne durch Konzepte einer anderen Domäne ersetzt. Dadurch sind andere Entwickler gezwungen ihr mentales Modell auf die neuen Konzepte anzupassen, wodurch die Komplexität des Quellcodes unnötig erhöht wird. Daher wird im Rahmen von Clean Code empfohlen, nur eine Bezeichnung pro Konzept zu verwenden und dieses auch beizubehalten. Ein Beispiel wäre dafür nicht die Begriffe `get`, `obtain`, `request` oder `receive` gleichzeitig zu verwenden, sondern sich für einen einheitlichen Begriff zu entscheiden. Natürlich muss auch an dieser Stelle der Kontext berücksichtigt werden, denn `get` und `request` können beispielsweise im Kontext von Webanwendungen wieder andere Bedeutungen haben.

Ein anderer Aspekt bei der Benennung ist das Vermeiden von Encodings. Dies führt zu einer ganzen Reihe von positiven Eigenschaften: Namen könnten über die Suche gefunden werden, sind leichter zu tippen und können in einem Fachgespräch richtig benannt werden, ohne dass es zu Problemen bei der Aussprache kommt.

Abseits von diesen Richtlinien existieren auch allgemeine Namenskonventionen, wie Camel Case (`myFunctionName`), Pascal Case (`MyFunctionName`) oder Kebab Case (`my-function-name`), welche einem die Bezeichnung erleichtern können. Programmiersprachen bringen zudem meist Konventionen für die Benennung eigener Konstrukte mit sich.

3.1.2 Methoden

Gute Methoden im Rahmen von Clean Code beginnen mit einem guten Namen. Dabei sollte es sich um ein Verb oder eine Verbphrase handeln. Sollte die Methode auf Daten zugreifen oder verändern, dann sollte sie mit dem Präfix `get` oder `set` versehen werden. Kommuniziert die Methode einen Zustand durch einen booleschen Wert, dann sollte das Präfix `is` verwendet werden.

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }
  return pageData.getHtml();
}
```

Listing 3.1: Beispiel für eine größere Methode aus [43]

Neben einer geeigneten Bezeichnung ist der Aufbau einer Methode in Clean Code ausschlaggebend. Demnach sind gute Methoden klein, sollten nur eine Aufgabe haben und nur ein Abstraktionsebenen besitzen. Abbildung 3.1 zeigt ein Beispiel für eine Methode, welche mehrere Aufgaben erfüllt und verschiedene Abstraktionsebenen nutzt. Diese Methode könnte so angepasst werden, das sie wie in Abbildung 3.2 aussieht. Dadurch wird sie übersichtlicher und verwendet Konzepte auf der gleichen Abstraktionsebene, wodurch der Sinn und Zweck der Methode besser nachzuvollziehen ist. Die Kombination von verschiedenen Abstraktionsebenen ist außerdem ein Indiz dafür, das eine Methode mehr als eine Aufgabe erledigt. Dies erhöht die Komplexität einer Methode zusätzlich. Außerdem sollten unterschiedliche Methoden nicht die gleichen Aufgaben umsetzen oder doppelt im Quellcode deklariert werden. Eine gute Richtlinie dafür ist das Prinzip „Don't Repeat Yourself (DRY)" [43]. Dies sorgt nicht nur dafür, dass weniger Quellcode geschrieben wird, sondern verringert auch die Komplexität.

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Listing 3.2: Beispiel für eine kleine Methode aus [43]

Bei dem Entwurf einer Methode kommt die Frage nach der Anzahl der Parameter auf. Allerdings lässt sich hier keine allgemeingültige Zahl definieren, da die Menge von dem Kontext abhängt. Nach Martin et al. gibt es allerdings ein paar einfache Richtlinien an denen man sich orientieren kann. Demnach hat eine Methode idealerweise keine Parameter. Ein oder zwei Methodenparameter sind vertretbar und jede Methode mit mehr als zwei Parametern sollte entweder vermieden oder gut begründet werden.

Diese Regeln basieren auf der Argumentation, dass jeder weitere Parameter die Komplexität einer Methode erhöht. So ist eine übermäßige Verwendung von Parametern oft ein Indiz dafür, dass eine Methode mehr als eine Aufgabe übernimmt und aufgrund der vielen Parameter schlecht zu testen ist. Natürlich gibt es hier auch ausnahmen und es sollte eine kontextabhängige Entscheidung getroffen werden.

Neben der Menge an Parametern ist auch die Art eines Parameters wichtig. Sollte eine Methode wirklich mehr als drei Parameter benötigen, so kann dies ein Indiz für neue Methoden oder eine neue Klasse sein, welche die Parameter zusammenfasst. So reduziert sich die Anzahl der Parameter und die Parameter erhalten zusätzliche semantische Beschreibung durch die Gruppierung in einer Klasse. Eine weitere Art mehrere Parameter zusammenzufassen ist die Verwendung von Listenparametern, sofern dies durch die Programmiersprache unterstützt wird.

Eine besondere Art von Parameter sind „flag arguments“ [43], welche es im Rahmen von Clean Code zu vermeiden gilt. Bei flag arguments werden boolesche Werte übergeben, welche ein klares Indiz dafür sind, dass die Methode mindestens zwei Aufgaben übernimmt. Abbildung 3.2 zeigt ein Beispiel für ein flag argument. Aus diesem Grund sollte die Änderung von Zuständen über flag arguments durch zusätzliche Methoden erfolgen. So könnte es sinnvoller sein das flag argument in Abbildung 3.2 zu eliminieren, in dem man eine weitere Methode schreibt.

Ein letztes Merkmal für gute Methoden ist die Command-Query-Separation. Abbildung 3.3 zeigt ein Beispiel für die Verletzung dieser Richtlinie. Dies führt zu Quellcode, in dem Werte innerhalb einer Bedingung gesetzt werden, da man die Rückgabe überprüfen möchte. Gute Methoden übernehmen nur eine Aufgabe und trennen daher diese beiden Aufgaben.

```
1  if (set("username", "unclebob"))...
```

Listing 3.3: Beispiel für eine Verletzung der Command Query Separation aus [43]

3.1.3 Klassen

Ähnlich wie bei Methoden beginnen gute Klassen mit einem guten Namen. Klassennamen und Objekte sollten durch ein Nomen oder eine Nominalphrase bezeichnet werden, um Entitäten zu beschreiben. Sie sollten keine Verben enthalten, da Klassen, im Gegensatz zu Methoden, keine Aktionen ausführen.

Eine Klasse besteht immer aus Methoden, daher ist eine gute Klasse eine Summe guter Methoden. Demnach sollten die Richtlinien für Klassen und Funktionen als gegenseitige Ergänzung gesehen werden. So gilt auch für Klassen die Richtlinie, dass sie möglichst klein sein sollten. Eine Klasse sollte außerdem ihre Implementierung verstecken und nicht durch ihre Methoden verfügbar machen. Einerseits kann eine Methode ihre Implementierung durch Zugriffsmodifikatoren, wie `public`,

`protected` oder *private* verstecken. Andererseits ist es wichtig die Zugriffe zu abstrahieren, damit andere Entwickler nicht einfach nur einen `getter` aufrufen um an die die Details zu kommen, sondern wirklich die Klasse und ihre Methoden nutzen.

Klassen nutzen Instanzvariablen um ihre Implementierung zu verstecken. Die Methoden einer Klasse manipulieren diese Variablen, um das Verhalten eines Objekts zu implementieren. In diesem Zusammenhang gibt es den Begriff „Kohäsion“, welcher beschreibt wie viele Methoden einer Klassen auf diese Variablen zugreifen. Eine Klasse ist maximal kohäsiv, wenn alle Methoden auf Instanzvariablen der Klasse zugreifen. Allerdings ist es oft nicht möglich solche Klassen zu implementieren, da einige Methoden diese Variablen vielleicht nicht brauchen. Allerdings bilden Klassen mit einer hohen Kohäsion ein logisches Ganzes, was für das Verständnis von Vorteil ist. Sollte dies nicht der Fall sein, dann könnte sich die Kohäsion durch auslagern von Variablen oder Methoden in andere Klassen erhöhen lassen.

3.1.4 Fehlerbehandlung

Die Behandlung von Fehlern ist ein wichtiger Teil der Softwareentwicklung und hat auch großen Einfluss auf die Qualität eines Programms. Gute Fehlerbehandlung beginnt mit der einfachsten und offensichtlichsten Grundlage: dem Schreiben guter Fehlermeldungen. Eine gute Fehlermeldung beschreibt nicht nur den konkreten Fehler, sondern gibt auch Informationen zum Kontext an. Beim Formulieren einer Fehlermeldung sollte immer die Sichtweise anderer Entwickler berücksichtigt werden. Diese haben nicht die gleiche Perspektive wie der ursprüngliche Entwickler und müssen die Fehlermeldung entsprechend interpretieren. Kurze und knappe Fehlermeldungen sind gut, solange sie die notwendigen Informationen liefern, um entsprechend handeln zu können.

```
1  public void delete(Page page) {
2    try {
3      deletePageAndAllReferences(page);
4    }
5    catch (Exception e) {
6      logError(e);
7    }
8  }
9
10 private void deletePageAndAllReferences(Page page) throws Exception {
11   deletePage(page);
12   registry.deleteReference(page.name);
13   configKeys.deleteKey(page.name.makeKey());
14 }
15
16 private void logError(Exception e) {
17   logger.log(e.getMessage());
18 }
```

Listing 3.4: Beispiel für Extraktion des try/catch Blocks aus [43]

Da die Behandlung von Fehlern auf Methodenebene erfolgt gibt es einige Richtlinien, welche die Richtlinien für Methoden aus Abschnitt 3.1.2 erweitern. So sollen demnach Methoden nur eine Aufgabe umsetzen, was an dieser Stelle auch auf Fehlerbehandlung zutrifft. Fehlerbehandlung ist eine Aufgabe und sollte demnach in eine eigene Methode ausgelagert werden.

Eine weitere negative Entscheidung mit weitreichenden Folgen ist es keine `null` Werte zurückzugeben. Dies sorgt dafür, dass die Nutzer einer Funktion bei jedem Methodenaufruf erst auf `null` überprüfen müssen. Sollte sich diese Entscheidung über mehrere Funktionen erstrecken. erhält man eine ganze Verschachtlung an Prüfungen. Diese Verschachtlungen lassen sich in Abbildung 3.5 erkennen.

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

Listing 3.5: Beispiel das Vermeiden von null Rückgabewerten [43]

Abbildung 3.6 zeigt eine weitere schlechte Entscheidung: die Verwendung von Fehlercodes. So wurden früher Fehlercodes verwendet, um dem Entwickler mitzuteilen, dass die Ausführung fehlgeschlagen ist. Dies führt allerdings dazu, dass der Rückgabewert immer überprüft werden muss. Zusätzlich muss dieser Fehler noch interpretiert werden, was entsprechendes Wissen über alle möglichen Fehlercodes erfordert. Exceptions vermeiden diese Notwendigkeit und sind daher vorzuziehen.

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else {
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

Listing 3.6: Beispiel für Error Codes aus [43]

Doch auch Exceptions können einige Schwächen aufweisen, besonders wenn es sich um selbst definierte Exceptions handelt. So sollte man sich als Entwickler immer Gedanken darüber machen, warum eine Exception auftritt und wie diese abgefangen wird. So kann es sinnvoll sein eine Exception zu schreiben, die mehrere Fehler abstrahiert. Eine hypothetische `FileIOException` könnte mehrere Fehler abfangen, wie falsche Pfade, falscher Dateiname, invalider Dateiname oder Fehler beim Lesen einer Datei. Dennoch könnte es sinnvoll sein diese nicht innerhalb einer allgemeinen Exception zu behandeln, da man diese vielleicht einzeln abfangen möchte. Die Abstraktion von Fehlern, besonders von Systemexceptions oder Exceptions von anderen Bibliotheken, gilt als gute Praxis. Durch die Implementierung eigener Exceptions entstehen weniger Abhängigkeiten gegenüber Quellcode von Dritten, wodurch der eigene Quellcode robuster wird, da er weniger anfällig gegenüber Änderungen ist.

3.1.5 Kommentare

Auch das Kommentieren von Programmcode ist ein wichtiges Merkmal für Clean Code. Grundsätzlich dienen Kommentare dazu, das Verständnis eines Programmes zu verbessern, indem bestimmte Abschnitte mit zusätzlichen Informationen versehen werden können. Im Rahmen von Clean Code werden Kommentare als Indiz für schlechten Code angesehen, da ultimativ der Quellcode ausreichen sollte, um den Ablauf zu verstehen. Laut den Autoren werden Kommentare allerdings oft genutzt, um schlechten Quellcode auszugleichen. Demnach sollten die Entwickler lieber die Zeit nutzen den Code zu verbessern, als Kommentare zu schreiben, welche den Quellcode nur unübersichtlicher machen. Dennoch gibt es Situationen, in denen Kommentare sinnvoll sind, wodurch eine allgemeingültige Vermeidung von Kommentaren nicht zu erwägen ist. Wie mit vielen Richtlinien bei Clean Code gilt es den Kontext abzuwägen und sich als Entwickler zu fragen, welche Informationen wirklich relevant sind.

Kommentare lassen sich außerdem in gute und schlechte Kommentare unterscheiden. Gute Kommentare folgen dabei Regeln, welche oft auch für gute technische Dokumente gelten. So sollten Kommentare kurz und knapp, aber informativ sein. Sie sollten die Absicht klar kommunizieren und mögliche negative Konsequenzen schildern. Im Gegensatz dazu sollten sie nicht um den Kern der Aussage herum reden oder irreführend sein. Sie sollten außerdem nicht redundant sein, zu viele Informationen enthalten oder gar als Positionsmarkierung innerhalb des Codes dienen.

Kommentare die Code auskommentieren, welcher im weiteren Verlauf der Entwicklung nicht mehr gebraucht wird, sollte ebenfalls entfernt werden. Andere Entwickler können im Verlauf der Zeit nicht unterscheiden, ob dieser Quellcode relevant ist.

3.2 Code Smells

Um Quellcode verbessern zu können muss man in der Lage sein dessen Probleme zu identifizieren. „Code Smells“ sind wiederkehrende Muster, welche auf ein tieferes Problem des Quellcodes hinweisen. Die Entscheidung, was genau ein Code Smells ist variiert nach Sprache oder Entwickler. Die im folgenden vorgestellten Code Smells gelten als allgemeingültig und basieren auf Fowler et al [23].

3.2.1 Bloaters

Bei „Bloaters“ handelt es sich um Code Smells, welche durch die Größe oder Menge der Verwendung negativ auffallen. Dabei kann es sich beispielsweise um Methoden oder Klassen handeln, welche so groß geworden sind, dass man mit ihnen nur noch schwierig arbeiten kann. Bloaters entstehen über die Zeit hinweg, sofern nicht entsprechende Gegenmaßnahmen getroffen werden.

Zwei offensichtliche Beispiele für Bloater sind lange Methoden (**Long Method**) und große Klassen (**Large Class**). Wie bereits in Abschnitt 3.1.2 und 3.1.3 beschrieben wird, sorgen diese Fälle schnell für Unübersichtlichkeit. Aus diesem Grund sollten geeignete Gegenmaßnamen getroffen werden, um Methoden oder Klassen zu verkleinern. Dies kann beispielsweise durch Dekomposition in mehrere Methoden oder Klassen geschehen.

Eine andere Art von Bloater ist die übermäßige Verwendung von elementaren Datentypen einer Programmiersprache (**Primitive Obsession**). Dies kann dazu führen, dass man unbewusst doppelten Quellcode schreibt, um diese Datentypen in verschiedenen Stellen einer Anwendung zu verwenden. Meist ist es sinnvoller diese in eine Klasse auszulagern und mit geeigneten Methoden zu versehen. Dadurch können sie wiederverwendet werden es entsteht kein doppelter Quellcode.

Lange Parameterlisten einer Methode sind eine weitere Art von Bloater (**Long Parameter List**). Lange Parameterlisten sind ein Indiz dafür, dass eine Methode mehr als eine Aufgabe implementiert und die Methode in mindestens zwei kleinere Methoden zerlegt werden sollte. Dieser Code Smells kann auch das Ergebnis eines Refactorings sein, bei dem Methoden generalisiert wurden und nun mehrere Aufgaben gleichzeitig erledigen.

Manchmal befinden sich an verschiedenen Stellen eines Programms die gleiche Kombination von Variablen (**Data Clumps**). Data Clumps entstehen in der Regel, wenn diese Variablen nicht zu Objekten zusammengefasst werden, um sie so wiederzuverwerten. Dies führt dazu, das sie wiederholt an verschiedenen Stellen im Quellcode deklariert werden müssen.

3.2.2 Object-Orientation Abusers

Bei „Object-Orientation Abusers“ handelt es sich um Code Smells, welche entstehen, wenn die Prinzipien objektorientierter Programmierung inkorrekt oder nicht richtig angewendet werden.

Ein Code Smells dieser Kategorie sind **Switch Statements**. Ein klassisches Beispiel für diesen Code Smells ist die Auswahl eines Algorithmus. Meist existieren Switch Statements nicht allein im Quellcode, sondern müssen an verschiedenen Stellen wiederholt implementiert werden. Sollten sich die Fälle nun ändern müssen diese überall angepasst werden. Eine bessere Lösung ist Switch Statements durch Polymorphie zu ersetzen. Dabei kann das komplette Statement durch eine Schnittstelle ersetzt werden und die einzelnen Fälle durch Kindklassen implementiert werden.

Temporäre Felder (**Temporary Fields**) sind Instanzvariablen, welche nur unter bestimmten Umständen einen Wert enthalten. Dies sorgt dafür, dass der Quellcode schwieriger zu verstehen ist, da meist nicht eindeutig ist, wann eine Variable einen bestimmten Wert einnimmt. Diese Art von Variable taucht bei Algorithmen auf, welche einen Wert für die Laufzeit des Algorithmus zwischenspeichern und sonst nicht weiter verwenden.

Vererbung ist ein hilfreiches Mittel bei der Entwicklung von Softwaresystemen, allerdings kann eine falsche Motivation zur Vererbung zu Problemen führen (**Refused Bequest**). Dieser Code Smells tritt auf, wenn jemand Quellcode wiederverwenden möchte, aber die Eltern- und Kindklasse komplett unterschiedlich sind und nur einen Teil der Methoden und Funktionalität der Elternklasse in der Kindklasse implementiert werden. Dies führt zu Problemen bei der Verwendung, da nicht eindeutig ist warum diese Klassen das Verhalten nicht korrekt implementieren.

Ein weiterer Code Smells ist die Implementierung identischer Funktionalitäten mit unterschiedlicher Schnittstellen (**Alternative Classes with Different Interfaces**). Dieser Code Smells tritt auf, wenn ein Entwickler eine bestimmte Funktionalität benötigt und nicht wusste, das diese existiert und im Anschluss an einer anderen Stelle neu implementiert.

3.2.3 Change Preventers

Die Code Smells der Kategorie„Change Preventers“ machen genau was der Name sagt: Änderungen am Programmcode verhindern. Dies führt dazu, das zusätzlicher Aufwand notwendig ist um die gewünschten Änderungen durchzuführen.

Sollten Änderungen an Methoden einer Klasse notwendig sein, welche nicht im Zusammenhang mit dem ursprünglichen Problem stehen, dann handelt es sich dabei um einen **Divergent Change**. Dieser Code Smells tritt auf, wenn Klassen mehrere Aufgaben übernehmen und diese innerhalb der Klasse zu stark gekoppelt sind. Um

diese Problematik zu vermeiden sollte die Klasse auf eine Aufgabe reduziert werden, indem die restliche Implementierung in eine oder mehrere Klassen ausgelagert wird.

Die **Shotgun Surgery** ist ähnlich wie ein Divergent Change. Doch anstelle von mehreren Änderungen an einer Klasse dreht es sich hierbei um eine Änderung an mehreren Klassen. Dies tritt auf, wenn eine Funktionalität, welche in einer Klasse zusammengefasst sein sollte, über mehrere Klassen verteilt ist.

Ein weiterer Code Smells ist die Existenz von parallelen Vererbungshierarchien (**Parallel Inheritance Hierarchies**). In diesen Fällen ist es notwendig, bei der Implementierung einer Kindklasse weitere Kindklassen zu implementieren, da die Hierarchien voneinander abhängen.

3.2.4 Dispensables

Bei „Dispensables“ handelt es sich um Konstrukte einer Programmiersprache, deren Abwesenheit keinen Einfluss auf den Quellcode haben und daher nicht existieren sollten.

Ein klassischer Code Smells dieser Kategorie sind Kommentare (**Comments**). Dieser Code Smells steht auch im direkten Bezug zu Clean Code und der Motivation, dass guter Quellcode als Dokumentation ausreichen sollte. Kommentare, die keinen zusätzlichen Informationsgehalt liefern, sollten vermieden werden.

Doppelter Programmcode (**Duplicate Code**) ist ein verbreiteter Code Smells. Dieses Problem tritt auf, wenn mehrere Entwickler parallel an verschiedenen Stellen eines Programms arbeiten oder schlichtweg Quellcode an verschiedene Stellen kopiert wird, um ähnliche Teilprobleme zu lösen. Dies führt zur Unübersichtlichkeit und fragilem Quellcode, das nun eine Änderungen an verschiedenen Stellen durchgeführt werden muss.

Im Verlauf eines Projekts sind meist Anpassungen ursprünglicher Entscheidungen notwendig, welche einen Einfluss auf die Klassenstruktur haben. So hat beispielsweise eine geplante große Klasse am Ende deutlich weniger Methoden. Dies könnte aber das Ergebnis eines Refactoring sein. Klassen (**Lazy Class**), die durch solche Änderungen keine wirkliche Existenzberechtigung mehr besitzen, sollten entfernt werden.

Datenklassen (**Data Class**), welche nur einige Variablen enthalten, sind oft ein Indiz dafür, dass andere Klassen zu viel Einfluss auf diese Klasse haben. Objekte sollten nicht nur als Ablageort für Daten fungieren. An dieser Stelle ist es sinnvoll Funktionalität der anderen Klassen in die Datenklassen zu übertragen, da das Verhalten eines Objektes in seiner Klasse definiert werden sollte.

Ein weiterer Code Smells, welcher problemlos entfernt werden kann, ist ungenutzter Quellcode (**Dead Code**).Dabei kann es sich um Variablen, Methoden, Klassen oder Schnittstellen handeln.

Mit der Zeit ändert sich Quellcode durch wechselnde Anforderungen. Eine Aufgabe des Entwicklers ist es, das Programm so zu strukturieren, dass Änderungen im späteren Verlauf möglich sind. Dies kann dazu führen, dass Quellcode umgesetzt wird, welcher noch nicht oder nie benötigt wird (**Speculative Generality**). Daher ist es meist eine gute Richtlinie nur Quellcode umzusetzen, welcher auch wirklich zum aktuellen Zeitpunkt benötigt wird. Dadurch kann man klar nachvollziehen, warum der Quellcode existiert.

3.2.5 Couplers

Die Kategorie „Couplers“ enthält Code Smells, welche bekannt für starke Kopplung sind und somit Änderungen am Quellcode behindern.

Die **Feature Envy** beschreibt das Verhalten einer Klasse, welche Methoden und Daten anderer Klassen mehr als die eigenen nutzt. Eine gute Richtlinie für objektorientiertes Design ist, dass Daten oder Methoden die sich zur gleichen Zeit ändern an einer Stelle im Programm sein sollten.

Inappropriate Intimacy beschreibt Klassen, welche interne Variablen oder Methoden anderer Klassen verwenden und dadurch zu stark gekoppelt sind. Gute Klassen sollten möglichst wenig voneinander wissen, damit diese wiederverwendet werden können. Um diesen Code Smells zu beheben sollten, sollten die Variablen oder Methoden in eine separate Klasse ausgelagert werden.

Die Verkettung von Methodenaufrufen bezeichnet man als **Message Chain**. Dieser Code Smells ist problematisch, weil das ursprüngliche Objekte an die Hierarchie der anderen Objekte gekoppelt ist. Dies macht es schwierig die Hierarchie zu ändern, da dann die Verkettungen ebenfalls angepasst werden müssen.

Klassen, welche nur eine Methode zur Verfügung stellen und die Implementierung an eine andere Klasse delegieren, bezeichnet man als **Middle Man**. Diese Klassen sollten entfernt werden, da sie nur die Komplexität erhöhen.

3.2.6 Weitere Arten von Code Smells

Ein weiterer Code Smells, welcher keiner konkreten Kategorie zugeordnet ist, ist das Fehlen von Funktionalitäten in Bibliotheksklassen (**Incomplete Library Class**). Um nicht jedes mal gewisse Funktionalitäten neu zu implementieren, baut ein Großteil des Programmcodes auf solchen Klassen auf. Allerdings ist es oft unmöglich diese Klassen anzupassen, wenn eine oder mehrere Methoden fehlen. Im schlimmsten Fall muss diese Klasse ausgetauscht werden. Eine andere Lösungen wäre die Implementierung einer Klasse, welche die Bibliotheksklasse abstrahiert und die zusätzliche Funktionalität implementiert. Eine weitere Lösung wäre die Implementierung einer Methode, welche die Bibliotheksklasse als Parameter erhält, um damit die gewünschte Implementierung zu realisieren.

3.3 Softwaredesign

Mit Clean Code und Code Smells wurde deutlich, das Quellcode sehr konkrete Qualitätsmerkmale besitzt. Allerdings entsteht guter Quellcode nicht in Isolation, sondern kann nur durch Verwendung validiert werden [42]. Außerdem existieren meist höhere Ziele wie Anpassbarkeit, Wartbarkeit oder Lose Kopplung, welche einen Einfluss auf den Quellcode haben. Um diese Ziele zu erreichen ist es erforderlich ein System in sinnvolle Objekte aufzuteilen, welches als der schwierigste Teil objektorientierter Programmierung gilt [24]. Im Laufe der Zeit wurden viele Praktiken entwickelt, die bei der Aufteilung helfen. Auch wenn diese Praktiken nicht explizit in der Untersuchung in Kapitel 5 berücksichtigt werden, so haben sie einen wichtigen Stellenwert und einen Einfluss auf die Datenquellen.

Eine dieser Praktiken ist „Programming to an Interface, not an Implementation“ [24]. Diese Praktik nutzt den Umstand aus, dass durch Vererbung nicht nur Funktionalität wiederverwendet werden kann, sondern auch Schnittstellen geteilt werden. Dadurch können Objekte ausschließlich aufgrund ihrer Schnittstelle behandelt werden. So muss ein Client den konkreten Typen von Objekten nicht kennen um diesen zu verwenden, wodurch Abhängigkeiten reduziert werden.

Eine weitere Praktik ist „Composition over Inheritance“ [24]. Dabei handelt es sich um zwei verschiedene Möglichkeiten Funktionalität innerhalb eines Systems wiederzuverwenden. Während bei der Vererbung die Funktionalitäten an die Kindklasse vererbt werden, wird bei der Komposition die Funktionalität in zusätzliche Klassen ausgelagert. Diese können dann je nach Bedarf wiederverwendet werden. Beide Techniken bieten ihre Vor- und Nachteile. Da Vererbung statisch definiert wird ist sie leicht zu implementieren. Durch Vererbung lässt sich auch eine Implementierung modifizieren, indem bestimmte Methoden überschrieben werden. Ein deutlicher Nachteil ist allerdings, dass die Klassen und ihre Vererbung nicht während der Laufzeit angepasst werden können. Aufgrund dessen hängen Kindklassen von ihrer Vererbung ab, was zu Abhängigkeiten führt. Außerdem wird dadurch Kapselung einer Klasse verletzt, da sie gezwungen wird sich zu ändern, wenn sich die Elternklasse ändert.

Sollte ein System hingegen durch Komposition mehrerer Objekte aufgebaut werden, kann die Implementierung zu jeder Zeit ausgetauscht werden. Da die Objekte über die Schnittstellen genutzt werden, bleibt auch die Kapselung intakt. Ein Nachteil ist allerdings, dass die Menge an Objekten stark zunimmt. Dies hat Einfluss auf die Performance und Komplexität der Anwendung. So hängt das Verhalten von den Beziehungen der Objekte untereinander ab und ist nicht mehr innerhalb einer Klasse definiert. Insgesamt bietet die Komposition viele Vorteile, welche die Wiederverwendung und Kapselung erhöhen. Aufgrund dessen sollte man Komposition gegenüber Vererbung vorziehen. Komposition wird durch „Delegation“ möglich, bei der die Implementierung einer Klasse oder Methode an andere Klassen delegiert wird [24]. Dies hat allerdings den Nachteil, dass die Implementierung stark parametrisiert ist.

3.4 Entwurfsmuster

Bei dem Entwurf einer Softwarearchitektur gibt es bestimmte Problemstellungen die wiederholt auftauchen. Für diese wiederkehrenden Probleme der objektorientierten Programmierung existieren bereits Lösungsansätze. Diese Lösungsansätze bezeichnet man auch als Entwurfsmuster [24]. Die bekanntesten Entwurfsmuster objektorientierter Programmierung sind die Muster nach Gamma et al. [24]. Darüber hinaus existieren allerdings viele weitere Entwurfs- und Architekturmuster [14] [19] [26]. Entwurfsmuster helfen Entwickler dabei weniger offensichtliche Abstraktionen für Probleme zu identifizieren und diese anzuwenden [24]. Die Entwurfsmuster nach Gamma et al. sind allerdings nicht allumfassend, sondern repräsentieren nur Lösungen für die häufigsten Problemstellungen [24]. Hinzu kommt, dass die Wahl der Programmiersprache einen Einfluss auf die möglichen Entwurfsmuster hat [24].

3.4.1 Kompositum

Ein Beispiel für ein Entwurfsmuster ist das Kompositum. Oft stehen Objekte auch in hierarchischen Beziehungen zueinander, welche sich nicht immer durch Vererbung abbilden lassen. Besonders bei der Wiederverwendung von hierarchischen Strukturen erweist sich Vererbung als das falsche Werkzeug. Ein Beispiel dafür sind Baumstrukturen, welche ein wiederkehrendes Konzept in der Informatik darstellen. Da Baumstrukturen in vielen Variationen existieren, ist es schwierig eine allgemeingültige Implementierung durch Vererbung zu realisieren. Das Kompositum Entwurfsmuster bietet für dieses Problem eine Lösung, indem es die Implementierung einer Teil-von Hierarchie ermöglicht. Dadurch lassen sich viele unterschiedliche Baumstrukturen implementieren, da deren Hierarchie sich ebenfalls als eine Menge von Teil-von Hierarchien darstellen lässt. Durch diesen Aufbau kann jedes beliebige Objekt verwendet werden, solange es die notwendigen Schnittstellen implementiert. Abbildung 3.1 veranschaulicht diese Struktur.

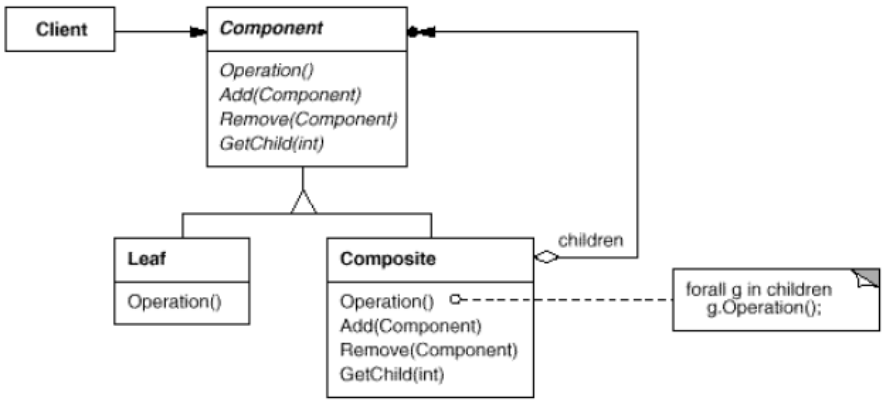

Abbildung 3.1: Das „Composite“ Entwurfsmuster nach Gamma et al. [24]

Dieses Entwurfsmuster sollte angewandt werden, wenn:

- Eine Teil-von Hierarchie abgebildet werden soll
- Einzelne oder Gruppen von Objekten gleich behandelt werden sollen

Da dieses Entwurfsmuster auf dem Konzept der Komposition basiert, übernimmt es dadurch auch die entsprechenden Vor- und Nachteile dieses Konzepts (siehe Abschnitt 3.3). Durch die Komposition ist nicht nur das Hinzufügen von neuen Komponenten einfacher geworden, sondern auch die Anwendung. So ist durch das Entwurfsmuster nicht mehr notwendig zwischen einzelnen Objekten oder einer Menge von Objekten zu unterscheiden, was in weniger Quellcode resultiert. Ein großer Nachteil ist allerdings, dass die Anwendung des Entwurfsmusters zu einem sehr allgemeinen Design führt. Auch wenn es jetzt einfacher ist neue Objekte hinzuzufügen ist es schwieriger nur eine bestimmte Art von Objekten zu verwenden. Dieses Beispiel für Entwurfsmuster macht den Einfluss auf Namen und Merkmale von Methoden deutlich.

3.4.2 Adapter

Bei der Arbeit mit objektorientierten Programmen kommt es häufig vor, dass eine konkrete Schnittstelle nicht zu der Schnittstelle eines anderen Objekts passt. Ein Beispiel dafür sind Klassen mit Schnittstellen, welche explizit für eine möglichst hohe Wiederverwendbarkeit entwickelt wurden. So kommt es vor, dass diese Schnittstelle nicht zu einer domänenspezifischen Schnittstelle passt.

Eine Lösung für dieses Problem ist das Entwurfsmuster Adapter. Dabei lässt sich die Problematik der unterschiedlichen Schnittstellen durch Vererbung oder durch Komposition lösen. Bei der Vererbung wird die erforderliche Schnittstelle durch eine neue Klasse geerbt, welche ebenfalls die Implementierung der inkompatiblen Schnittstelle erbt. Dadurch können die Methoden der erforderlichen Schnittstelle mit Hilfe der geerbten Methoden der inkompatiblen Schnittstelle realisiert werden. Bei der Komposition hingegen erbt eine neue Klasse nur die erforderliche Schnittstelle und delegiert die Implementierung der Methoden an ein Objekt der inkompatiblen Schnittstelle. Abbildung 3.2 zeigt das Entwurfsmuster mit der Lösung durch Vererbung.

Dieses Entwurfsmuster sollte angewandt werden, wenn:

- Eine bereits vorhandene Klasse verwendet werden soll, welche aber nicht über eine kompatible Schnittstelle verfügt
- Man eine wiederverwendbare Klasse möchte, welche die Verwendung von Klassen mit inkompatiblen Schnittstellen erlaubt

- Man eine Menge von verschiedenen Objekten mit inkompatiblen Schnittstellen hat und eine Vererbung der einzelnen Klassen nicht sinnvoll ist (nur bei Komposition)

Die Verwendung dieses Entwurfsmusters geht mit einigen Konsequenzen einher. Sollte der Adapter mit Hilfe von Vererbung realisiert werden, dann kann nur die konkrete, adaptiere Klasse verwendet werden. Eine Adaptierung der Kindklassen ist nicht möglich. Allerdings kann durch die Vererbung das Verhalten der adaptieren Klasse angepasst werden, da entsprechende Methoden in der Adapterklasse überschrieben werden können. Ein Adapter durch Komposition kann hingegen mit der adaptieren Klasse, sowie dessen Kindklassen arbeiten. Im Gegenzug gestaltet sich allerdings das Anpassen des Verhaltens schwieriger, da sich die Methoden der adaptieren Klasse nicht mehr überschreiben lassen. Um dies zu erreichen, ist eine weitere Klasse notwendig, welche die Implementierung der zu adaptierten Klasse erbt, überschreibt und dann von dem Adapter verwendet wird.

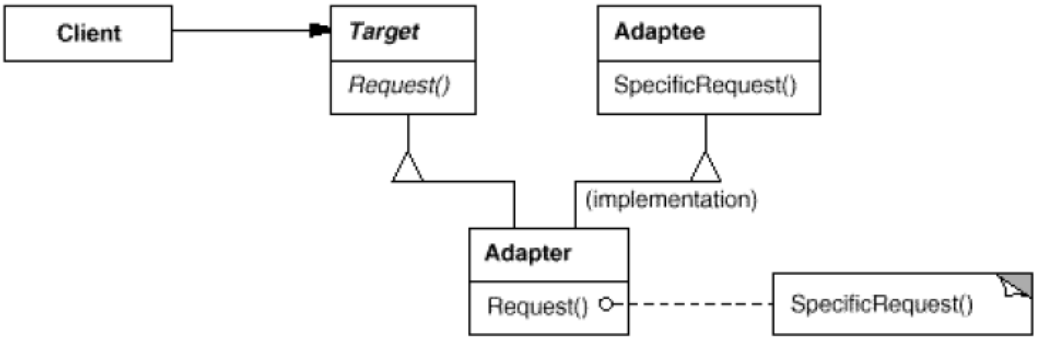

Abbildung 3.2: Das „Adapter“ Entwurfsmuster nach Gamma et al. [24]

3.4.3 Beobachter

Aufgrund der Aufspaltung eines Programms in verschiedene Objekte ist es notwendig die verschiedenen Daten konsistent zu halten. Daher muss eine Beziehung zwischen Objekten geschaffen werden, welche den Zustand der relevanten Objekte kommuniziert. Allerdings gilt es dabei eine zu enge Kopplung zu vermeiden, da dadurch die Wiederverwendbarkeit der einzelnen Objekte beeinträchtigt wird.

Diese Art der Beziehung kann mit dem Beobachter Entwurfsmuster realisiert werden. Dabei wird eine Beziehung modelliert, welche den Zustand eines Objekts an ein anderes kommuniziert, wenn sich dieses ändert. Dabei teilen sich die Objekte in Subjekte und Beobachter auf. Ein Subjekt kann dabei eine beliebige Anzahl an Beobachtern besitzen. Sollte sich der Zustand eines Objekts ändert, werden alle Beobachter benachrichtigt, welche im Anschluss den neuen Zustand des Subjektes erfragen können. Das Subjekt kennt dabei seine Beobachter nicht direkt, wodurch

eine möglichst lose Kopplung gewährleistet wird. Abbildung 3.3 zeigt die notwendige Objektstruktur für eine solche Interaktion.

Dieses Entwurfsmuster sollte angewandt werden, wenn:

- Eine Abstraktion zwei verschiedene Aspekte realisiert, welche voneinander abhängen, aber durch unterschiedliche Objekte implementiert werden
- Die Änderung eines Objekts die Änderung anderer Objekte erfordert
- Andere Objekte benachrichtigt werden sollen ohne die konkrete Implementierung zu kennen

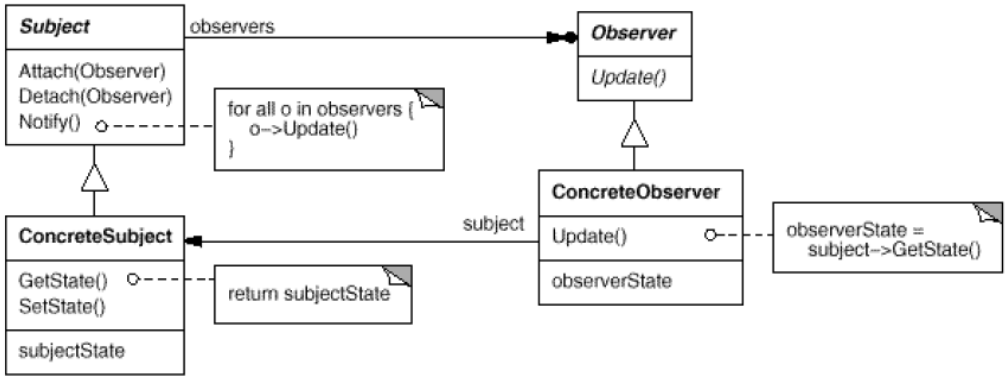

Abbildung 3.3: Das Beobachter Entwurfsmuster nach Gamma et al. [24]

Die wichtigste Konsequenz des Beobachter Entwurfsmusters ist die lose Kopplung zwischen Subjekt und Beobachter. Durch die Nutzung der Observer Schnittstelle kennt ein Subjekt seine konkreten Beobachter nicht. Durch dieses Entwurfsmuster ist auch eine Kommunikation über die verschiedenen Ebenen einer Anwendung möglich, ohne die Abstraktion zwischen den Ebenen zu kompromittieren. Außerdem ist durch das Entwurfsmuster eine Broadcastkommunikation möglich, da Observer nach belieben hinzugefügt oder entfernt werden können. Ein großer Nachteil ist jedoch, dass die Beobachter unerwünschte oder unerwartete Updates erhalten können. Da die Beobachter keine Kenntnis voneinander besitzen, erhöht dies die Performancekosten. Außerdem kann eine übermäßige Verwendung des Entwurfsmustern zu einer Kaskade von Änderungen führen, welche unter Umständen nicht mehr nachvollziehbar ist. Dies erhöht die Komplexität und sorgt dafür, das Fehler nicht mehr eindeutig nachvollziehbar sind.

3.5 Design Prinzipien

Neben erprobten Lösungsansätzen, wie den Entwurfsmustern, existieren Designprinzipien für gute Softwarearchitekturen. Ein bekanntes Beispiel für solche Prinzipien sind die S.O.L.I.D Prinzipien [42]. Die Prinzipien wurden im Jahr 2000 von Robert Martin vorgestellt [42]. In den folgenden Abschnitten werden die einzelnen Prinzipien kurz beschrieben.

3.5.1 Single-Responsibility Principle

Das Single-Responsibility Principle (SRP) sagt aus, dass eine Klasse nur einen Grund zur Veränderung besitzen sollte [42]. Dieses Prinzip wurde ursprünglich von Tom DeMarco [16] und Meilir Page-Jones [49] beschrieben und von Robert Martin aufgegriffen [42]. DeMarco und Page-Jones bezeichnen dieses Prinzip als Kohäsion. Sollte eine Klasse mehr als einen Grund zur Veränderung besitzen, führt sie mehrere Aufgaben aus. Dies ist ebenfalls ein negatives Merkmal von Clean Code. Durch mehrere Aufgaben in einer Klasse wird der Quellcode gekoppelt und es wird schwierig Änderungen an nur einer Aufgabe durchzuführen. Wenn allerdings eine Klasse nur eine Aufgabe besitzt, dann können keine unerwarteten Nebeneffekte auftreten.

3.5.2 Open-Closed Principle

Das Open-Closed Principle (OCP) geht auf Bertrand Meyer [46] zurück und wird von Robert Martin wie folgt beschrieben: „Software entities (classes, functions, etc) should be open for extension, but closed for modification“ [42]. In modernen objektorientierten Programmiersprachen wird dies durch Abstraktion erreicht [42]. Dadurch ist es zu jeder Zeit möglich die interne Implementierung einer Klasse anzupassen, ohne dass dies einen Effekt auf Quellcode außerhalb der Klasse hat. Andere Klassen können nur über die Abstraktion dieser Klasse kommunizieren, wodurch sich die interne Implementierung nicht von außen modifizieren lässt. Allerdings ist die Einhaltung des OCP mit Kosten verbunden, da es Zeit und Aufwand kostet gute Abstraktionen zu implementieren [42].

3.5.3 Liskov Substitution Principle

Das Liskov-Substitution Principle wurde ursprünglich von Barabara Liskov [39] beschrieben und wird von Robert Martin wie folgt definiert: „Subtypes must be substitutable for their base types“ [42]. Das Ziel des Liskov-Substitution Principle ist es, dass man zu jedem gegebenen Zeitpunkt den Typ einer Klasse durch den Typ ihrer Elternklasse ersetzen kann. Durch dieses Prinzip wird Quellcode wartbarer, wiederverwendbarer und robuster [42]. Dies bedeutet, dass eine Klasse alle Methoden ihrer Schnittstellen korrekt implementieren muss. Eine Möglichkeit dies zu erreichen ist durch „Design by Contract“ [45].

3.5.4 Interface Segregation Principle

Das Interface Segregation Principle sagt aus, dass Klassen nicht gezwungen werden sollte Methoden zu implementieren, welche sie nicht benötigen [42]. Dies passiert, wenn Schnittstellen zu viele Aufgaben übernehmen und so groß werden, dass Kindklassen nur einen Teil der Methoden implementieren. Eine Möglichkeit, um Verantwortlichkeiten abzugeben, ist das Aufteilen einer Schnittstelle in mehrere kleinere Schnittstellen [42]. Die einzelnen Schnittstellen übernehmen dann kleinere und abgegrenzte Aufgaben. Eine andere Möglichkeit ist es Funktionalitäten zu delegieren, sodass diese aus der ursprünglichen Schnittstelle entfernt werden können [42].

3.5.5 Dependency-Inversion Principle

Quellcode wird oft nicht in Isolation geschrieben und hängt mindestens von der API der Programmiersprache und den selbst definieren Objekten ab [42]. Sollten Änderungen erforderlich sein, so kann es schwierig werden diese durchzuführen, wenn die Abhängigkeiten einer Klasse nicht dynamisch angepasst werden können [42]. Hinter dieser Problematik steht das Dependency-Inversion Principle (DIP) und versucht dies wie folgt zu lösen [42]

- „High-level modules should not depend on low-level modules. Both should depend on abstractions“
- „Abstractions should not depend on details. Details should depend on abstractions“

Ein Großteil von Programmen besteht aus mehreren abstrakten Schichten. Diese sollten nicht abhängig voneinander sein und durch Abstraktionen miteinander kommunizieren [42]. Sollten sie allerdings auf konkreten Details basieren entstehen Abhängigkeiten, welche sich schwierig anpassen lassen [42]. Dies lässt sich vermeiden, wenn die Implementierung als Parameter eines Konstruktors oder einer Methode übergeben wird. Dadurch lässt sich die Abhängigkeit jederzeit austauschen.

3.6 Test Driven Development

Bei Test-Driven-Development (TDD) handelt es sich um einen Entwicklungsstil, bei dem die Entwicklung der Software durch automatische Tests vorangetrieben wird [34]. Test-Driven-Development geht auf Kent Beck zurück und setzt voraus, dass nur dann neuer Quellcode geschrieben wird, wenn bereits ein fehlgeschlagender Test existiert [34]. Durch diese Voraussetzung wird man als Entwickler gezwungen, sich klare Gedanken darüber zu machen, was und wie es implementiert werden muss [34]. Durch diesen Entwicklungsstil wird ein Prozess geschaffen, welcher zu einer klaren Akzeptanz oder Ablehnung von Programmcode führt [34].

Der Prozess lässt sich dabei durch die folgenden fünf Schritte beschreiben [34]. Als Erstes wird ein neuer Test geschrieben, welcher die zu implementierende Funktionalität testet. Im Anschluss werden alle Tests ausgeführt, um sicherzustellen, dass der neue Test fehlschlägt. Sobald dies bestätigt ist, können Änderungen am Quellcode vorgenommen werden, um die neue Funktionalität zu implementieren. Im Anschluss werden die Tests erneut ausgeführt, um zu kontrollieren, ob alle Tests bestehen. Sollte dies nicht der Fall sein, werden solange Änderungen durchgeführt bis alle Tests bestehen. Zum Ende wird der Quellcode verbessert, indem potentiell entstandener doppelter Quellcode entfernt wird. Durch automatische Softwaretests erhalten Entwickler die Sicherheit, dass eine oder mehrere Änderungen am Quellcode keine neuen Fehler hervorbringen [34]. Dies nimmt Entwicklern die Angst weitere Änderungen durchzuführen, da man sich jederzeit sicher sein kann, dass die Software funktioniert [34]. Die Sicherheit wird vor allem durch Unit Tests garantiert [34]. Unit Tests überprüfen die kleinsten Einheiten von Quellcode, um sicherzustellen, dass die ganze Software funktioniert [34]. Außerdem sollten Tests den gleichen Stellenwert wie der Quellcode der zu testenden Anwendung einnehmen [43]. Das soll verhindern, dass Tests vernachlässigt werden und so ihren ursprünglichen Nutzen verlieren [43]. Kent Beck hat außerdem einige Prinzipien definiert, welche bei der Entwicklung guter Tests helfen sollen [34].

Das erste wichtige Prinzip heißt „Test First“ [34]. Dies gibt eine klare Antwort darauf, wann ein Test geschrieben werden sollte. Dies ist dadurch motiviert, dass wenn die Software einmal geschrieben ist und grundsätzlich funktioniert, keine automatischen Tests mehr geschrieben werden. Durch diesen Ansatz wird außerdem der beschriebene Stress des Entwicklers reduziert, da er zu jeder Zeit Feedback zur Funktionalität der Software erhält.

Das Prinzip „Isolated Tests“ [34] sagt aus, dass Tests keinen Einfluss aufeinander haben sollten. Dies ist oft bei komplexen Anwendungen mit Zuständen nicht einfach und erfordert die Zerlegung des Problems in kleine testbare Schritte. Der Vorteil ist, dass man daraus viele kleine entkoppelte und hoch kohäsive Objekte erhält, welche ein klares Ziel von Clean Code sind.

„Assert First“ [34] hat einen ähnlichen Hintergrund wie das Prinzip „Test First“. Da man zu Beginn ausreichend über die Prüfungsbedingungen nachdenkt, ist man ebenfalls automatisch dazu gezwungen, sich Gedanken über die Architektur der Software, potentielle Namen, Typen oder weitere Tests zu machen.

Tests werden von Menschen für Menschen geschrieben. Aus diesem Grund müssen nach Kent Beck ebenfalls „clean“ sein, was vor allem durch Lesbarkeit garantiert werden soll [34]. Auch hier kann Clean Code durch Richtlinien wie sinnvolle und prägnante Namen, sowie kleine Funktionen bzw. Tests helfen. Zudem beschreibt Kent Beck, dass Tests über aussagekräftige Daten verfügen sollen. So sollen Tests die erwarteten und erhaltenen Daten klar beschreiben und die Beziehung zwischen den Daten deutlich machen.

3.7 Fazit

Diese Literaturrecherche macht deutlich, dass die Qualität von Quellcode vielen Einflussfaktoren unterliegt. Auch wenn Sammlungen von Richtlinien, wie Clean Code, einen gute Hilfe bieten, so kann diese nie allgemeingültig sein. Hinzu kommt, dass die hier gezeigten Merkmale nur für den Entwurf von Methoden erarbeitete wurden. Dennoch lassen sich mit den hier beschriebenen Merkmalen deutliche Unterschiede zwischen „gutem“ und „schlechtem“ Quellcode beschreiben.

Ein einfaches Regelwerk, wie Clean Code, macht den Wert von Iteration deutlich. Codequalität ist nicht etwas, was man im ersten Moment produziert. Stattdessen helfen einem diese Regeln ein Rahmenwerk zu schaffen, an dem sich ein Entwickler orientieren kann. Dieser Aspekt spiegelt sich auch in Code Reviews wieder, welche ebenfalls nicht von einem perfekten Ergebnis im ersten Versuch ausgehen.

Bei der Betrachtung der einzelnen Richtlinien, Prinzipien und Architekturentscheidungen wird deutlich, wie groß der Einfluss einiger weniger Methoden sein kann. Starke Kopplung, Fehlerbehandlung und Message Chains sind nur einige Beispiele dafür. Auf der anderen Seiten wird allerdings auch deutlich, dass die Verwendung von Prinzipien und Mustern ebenfalls einen starken Einfluss auf Methoden haben. So hat es einen gewissen Wert die Methoden nach Vorgabe eines Entwurfsmuster zu benennen, auch wenn diese im Konflikt zu einem Regelwerk stehen sollten. Dies hebt ebenfalls die Notwendigkeit hervor, dass ein Entwickler diese Muster und Prinzipien kennen und verstehen muss, um sie richtig einzuordnen. Dies bestätigt das Problem von Code Reviews aus Abschnitt 2.1.

Zuletzt gilt es noch einen wichtigen Punkt festzuhalten. Alle Richtlinien, Prinzipien, Muster oder Entscheidungen haben oft keinen Effekt auf das eigentliche Ergebnis des Quellcodes. Ob der Quellcode gut oder schlecht ist, dass Ziel und Ergebnis bleibt das Gleiche, sofern keine Fehler entfernt werden. Die Vorteile von gutem Quellcode werden erst in späteren Schritten deutlich, was allerdings nicht bedeutet, dass die Effekte zu vernachlässigen sind. Dies bestätigt auch nochmal das Bedürfnis nach Software, welche die Entwicklung von Quellcode auf semantischer Ebene unterstützt. Die Motivation für probabilistische Quellcodemodelle aus Abschnitt 2.3 könnte durch Deep Learning diese Lücke schließen.

4 Deep Learning

Deep Learning ist ein Teilgebiet von Machine Learning [50] und befasst sich mit Algorithmen und statistischen Modellen, um Problemstellungen durch Inferenz zu lösen, ohne dabei konkrete Handlungsweisen vorzugeben. Zentraler Untersuchungsgegenstand von Deep Learning sind Neuronale Netze. Der theoretische Hintergrund von Neuronalen Netzen basiert auf einer groben Analogie zum menschlichen Gehirn, welches aus einem biologischen Netz mit ungefähr 86 Milliarden Neuronen besteht [50]. Diese beiden Konzepte werden meist aus Sicht der Signalverarbeitung betrachtet [17]. Das menschliche Gehirn bildet ein biologisches Netz und verwendet elektrische und chemische Signale, um Information zu verarbeiten [50]. Ein Neuronales Netz versucht diesen Ansatz durch ein mathematisches Modell zu realisieren.

In den letzten Jahrzehnten wurden die relevanten Merkmale, sogenannte Features, manuell erarbeitet [37]. Dabei wurden die relevanten Datensätze auf geeignete Features untersucht, um diese in ein Format zu bringen, welches den Machine Learning Prozess möglichst effektiv unterstützt [50]. Ein Neuronales Netz hingegen entscheidet selbständig welche Features relevant sind, um die gegebenen Daten zu beschreiben [50]. Dies wird auch als Representation Learning bezeichnet [37]. Die Tatsache, dass Neuronale Netze in der Lage sind auf Basis einer ausreichend großen Datenmenge zu lernen, ist ein wichtiger Grund für deren Erfolg [37]. Allerdings gibt es dabei keinen allgemeingültigen Ansatz und jedes Neuronale Netz konstruiert seine Features unterschiedlich [50].

Neuronale Netze lassen sich durch zwei Arten des Lernens unterscheiden: Supervised Learning und Unsupervised Learning [17]. Supervised Learning extrahiert die relvanten Features aus vorher klassifizierten Daten [17]. Unsupervised Learning versucht relevante Features selbstständig aus nicht klassifizierten Daten zu gewinnen [17]. Die häufigste Form von Machine Learning ist Supervised Learning [37]. Darüber hinaus existieren auch hybride Formen [17], welche versuchen diese Ansätze zu kombinieren, um bessere Ergebnisse zu erreichen.

4.1 Neuronen

Die zentrale Einheit eines Neuronalen Netzes ist ein künstliches Neuron [50]. Neuronen werden durch ihre Ein- und Ausgabe definiert. Die Eingabe eines Neurons kann binär oder diskret sein, die Art der Ausgabe hängt von der Wahl der Aktivierungsfunktion ab [50]. Abbildung 4.1 zeigt den Aufbau eines künstlichen Neurons [50].

Ein Neuron nimmt eine Menge von Eingabewerten x_i entgegen, welche mit einem Gewicht w_{ij} multipliziert werden [50]. Diese Werte werden im Anschluss aufaddiert und an eine Aktivierungsfunktion übergeben [50]. Anhand des Eingabewertes entscheidet die Aktivierungsfunktion, ob das Neuron aktiviert wird, was in Relation zur Analogie des menschlichen Gehirns steht [50].

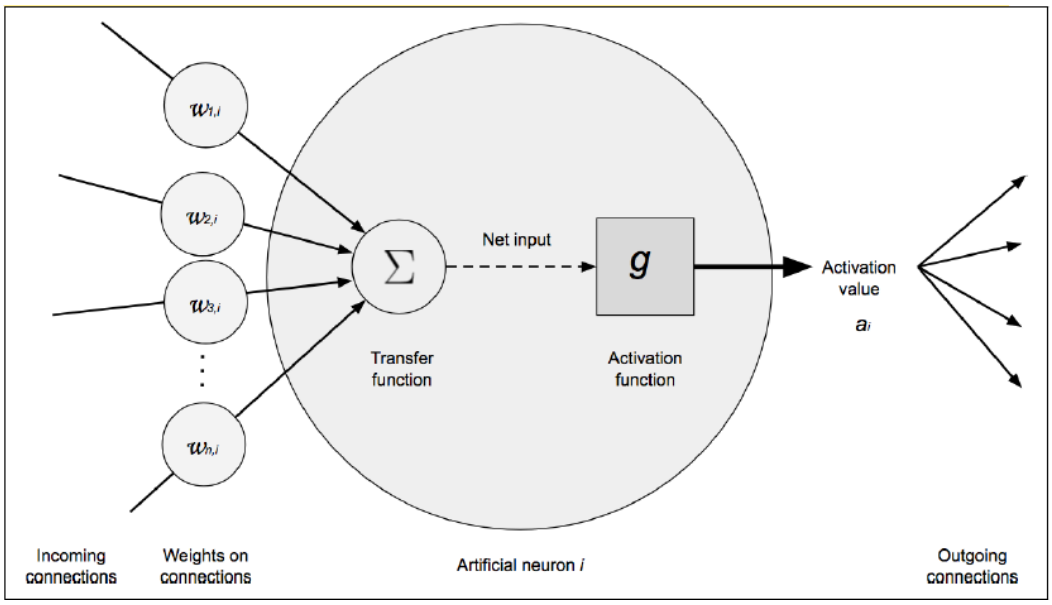

Abbildung 4.1: Der Aufbau eines künstlichen Neurons aus [50]

Zusammengefasst lässt sich dies als eine Funktion ausdrücken, welche in Gleichung (4.1) zu sehen ist [50]. Die Gewichte und Eingaben werden dabei als Vektor W_i und A_i zusammengefasst, um ein Skalarprodukt zu bilden [50]. Im Anschluss wird noch ein *Bias* addiert [50]. Dabei handelt es sich um einen Skalar, welcher sicherstellt, dass in einem Neuronalen Netze mindestens einige Neuronen aktiviert werden, auch wenn die Eingaben nicht ausreichen [50]. Das Ergebnis wird an die Aktivierungsfunktion g weitergereicht, welche die Ausgabe des Neurons bestimmt [50].

$$a_i = g(W_i \cdot A_i + b) \tag{4.1}$$

Der Aufbau eines künstlichen Neurons geht auf das sogenannte Perzeptron [54] zurück, einem linearen Modell für binäre Klassifikation [50]. Dessen Vorläufer ist die Threshold Logic Unit von McCulloch und Pitts [44] [50]. Das Perzeptron wird durch eine Treppenfunktion aktiviert, wie sie in Gleichung (4.2) zu sehen ist. Sollte der Eingabewert der Treppenfunktion den festgelegten Schwellwert θ_j überschreiten, dann wird das Neuron aktiviert. Bei Perzeptrons handelt es sich dabei überweise um die Heaviside Funktion mit einem Schwellwert von 0,5. Abhängig von der Eingabe ist das Ergebnis ein binärer Wert (0 oder 1) [50].

Das Konzept des Lernens wird in Perzeptrons durch die Gewichte der Eingabewerte modelliert. Sie können entweder manuell oder durch einen Algorithmus angepasst werden, um die Klassifikation zu optimieren [50]. Durch die Gewichte wird entschieden, welche Eingabewerte relevanter sind als andere.

$$a_i = \begin{cases} 0 & if \sum_{i=1}^{n} w_i x_i \leq \theta_j \\ 1 & if \sum_{i=1}^{n} w_i x_i > \theta_j \end{cases} \tag{4.2}$$

Das Perzeptron kann allerdings nur eine limitierte Menge von Mustern erkennen [50]. So wurde die nicht-Lösbarkeit von nicht-linearen Problemen durch Perzeptrons als Versagen angesehen [50]. Dies führte zu mangelnden Interesse an Neuronalen Netzen bis in die 1980er Jahre [50].

4.2 Multilayer Feed-Forward Netzwerke

Im späteren Verlauf zeigte sich, dass Perzeptrons doch in der Lage waren nicht-lineare Probleme zu lösen, wenn man mehrere Perzeptrons kombiniert [50]. Dies führte im weiteren Verlauf zu sogenannten Multilayer Feed-Forward Netzwerken [50]. Ein Multilayer Feed-Forward Netzwerk ist ein Neuronales Netz, welches aus mehreren Ebenen künstlicher Neuronen besteht [50]. Ein Beispiel ist in Abbildung 4.2 zu sehen. Das Netzwerk besteht dabei aus mehreren Ebenen parallel angeordneter Neuronen, deren Ausgabewerte zu den Eingabewerten der folgenden Neuronen werden [50]. Die Ebenen unterscheiden sich wie folgt [50]:

Input layer: Die erste Ebene eines Feed-Forward Netzwerks ist die Eingabeebene, welche die Eingabewerte an das Netzwerk weitergibt [50]. Üblicherweise besitzt eine Eingabeebene so viele Neuronen, wie der Eingabevektor lang ist [50]. In klassischen Feed-Forward Netzwerken sind alle Neuronen der Eingabeschicht mit der darauf folgenden Schicht verbunden [50].

Hidden layer: Auf die Eingabeebene folgt mindestens eine oder beliebig viele versteckte Ebenen [50]. Die Ausgabe der einzelnen Ebenen (inklusive der Eingabeschicht) wird an die einzelnen versteckten Schichten weitergegeben [50]. Die Übertragung der Werte wird auch als „forward propagation“ bezeichnet [50]. Die versteckten Ebenen sind der Grund, warum Neuronale Netze in der Lage sind, nicht-linear Probleme zu lösen [50].

Output layer: Die Ausgabeschicht besteht aus einer Reihe von Neuronen, welche in Abhängigkeit des gewünschten Ergebnisses modelliert werden [50]. Über diese Ebene erhält man das finale Ergebnis des Netzwerks [50]. Dabei kann es sich um einen reellen Ausgabewert handeln oder eine Menge von Wahrscheinlichkeiten [50].

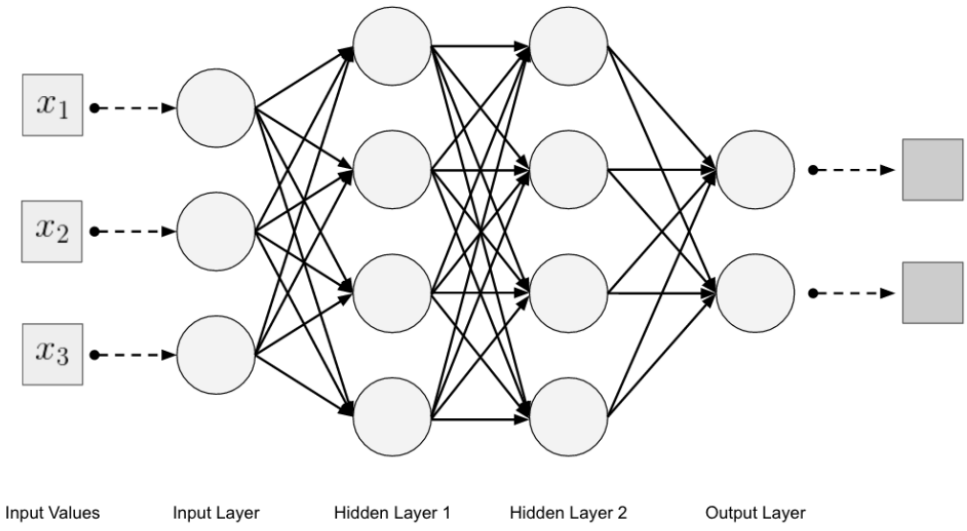

Abbildung 4.2: Der Aufbau eines Multilayer Feed-Forward Netzwerks aus [50]

Die einzelnen Ergebnisse der Neuronen, welche an die nächste Ebene weitergegeben werden, nennt man Aktivierungen [50]. Die Aktivierungen, welche durch die Aktivierungsfunktionen erzeugt werden, steuern das Verhalten eines Neuronalen Netzes [50]. Die derzeit populärste Aktivierungsfunktion ist die Rectified Linear Unit (ReLU) in Gleichung (4.3) [37] [50].

$$a_i = max(0, x) \tag{4.3}$$

Um eine binäre Klassifikation mit einem Multilayer-Feed-Forward Netzwerk durchzuführen nutzt die Ausgabeebene die Sigmoidfunktion [50]. Die Ausgabe der Sigmoidfunktion hat einen Wert zwischen 0 und 1. Die Funktion ist in Gleichung (4.4) dargestellt.

$$a_i = \frac{1}{1 + e^{-z}} \tag{4.4}$$

Wie in Abschnitt 4.2 beschrieben wurde, lernt ein einzelnes Perzeptron durch die Anpassung der Gewichte. Multilayer-Feed-Forward Netzwerke besitzen Gewichte und Biases, welche die Relevanz der einzelnen Aktivierungen entweder verstärken oder abschwächen können [50]. Um die Gewichte mit einem Algorithmus anpassen zu können, muss ein Mittel gefunden werden, womit der Erfolg des Netzwerks gemessen und basierend darauf die Gewichte und Biases angepasst werden können [50]. Ein verbreiterter Ansatz für Lernen in Neuronalen Netzen ist *Backpropagation*.

4.3 Backpropagation

Backpropagation ähnelt dem Lernansatz von Perzeptrons [50]. Dabei handelt es sich um einen pragmatischen Ansatz, um den Fehler eines Netzwerks auf dessen Gewichte zu verteilen [50]. Bei Backpropagation wird zu Beginn das Ergebnis einer Eingabe bestimmt [50]. Sollte das Ergebnis mit der vorher definierten Klasse der Eingabe übereinstimmen, dann ändert sich nichts [50]. Sollte allerdings das Ergebnis abweichen, dann werden die Gewichte und Biases des Netzwerks angepasst [50]. Backpropagation wurde schon in den 1970er Jahren erfunden, aber die Relevanz für Machine Learning erst 1986 durch ein Paper von Rumelhart et al. erkannt [55].

Input: network parameters $\mathbf{w}$, loss function L, training data $\mathbf{D}$, learning rate $\alpha > 0$
while *termination conditions are not met* **do**
  (features, labels) ← D.getRandomMiniBatch()
  out ← getNetworkOutput(w, features)
  $\frac{\partial L}{\partial \mathbf{w}}$ ← calculateParameterGradients($\mathbf{w}$, L, out, labels)
  w ← $w - \alpha \frac{\partial L}{\partial \mathbf{w}}$
end

Algorithm 1: Backpropagation mit Stochastic Gradient Descent aus [50]

Das Ziel von Backpropagation ist es, den Fehler zwischen den erwarteten und eigentlichen Ergebnis zu minimieren [50]. Dieser Fehler wird durch eine Loss Funktion bestimmt [50]. Durch das Minimieren dieser Funktion verringert sich der Fehler des Neuronalen Netzes [50]. Eine Möglichkeit, die Loss Funktion zu minimieren, ist mit Stochastic Gradient Descent [50]. Der Gradient soll den Weg zu einem Minimum der Loss Funktion weisen, welcher dann als Parameter genutzt werden kann [50].

Der Backpropagation Algorithmus erhält drei Parameter als Eingabe: Das Neuronale Netz, die Trainingsdaten und einen weiteren Parameter, der als „Learning Rate“ bezeichnet wird. Die Learning Rate beschreibt, wie stark die Gewichte während eines Durchlaufs des Algorithmus angepasst werden dürfen [50]. Sollte der Wert zu groß sein, dann könnte der richtige Wert der Gewichte, für ein optimales Ergebnis, verfehlt werden [50]. Ist der Wert zu klein, dann kann der Trainingsprozess sehr lange dauern, bis es zu einem guten Ergebnis kommt [50]. Im Anschluss läuft der Algorithmus über die verschiedenen Trainingsdaten. Dies hält so lange an bis eine Bedingung erreicht ist, die das Training beendet [50]. Ein Beispiel dafür ist die maximale Anzahl der Epochen, ein Parameter der die Anzahl der Trainingsiterationen beschreibt. Abbildung 1 zeigt den Pseudocode zu diesem Vorgehen.

Um die Fehler eines Neuronalen Netzes zu berechnen wird eine passende Loss Funktion benötigt. Da der Gradient der Funktion ermittelt werden soll muss diese Funktion ableitbar sein [50]. Ein Beispiel für eine Loss Funktion, welche für binäre Klassifikation verwendet wird, ist in Gleichung (4.5) zu sehen [50]. Diese Funktion ist als „negative log likelihood“ oder „binary cross entropy“ bekannt [50]. Dabei wird der

durchschnittliche Fehler der logarithmischen Wahrscheinlichkeiten berechnet [50]. W und b bezeichnen die Gewichte und entsprechenden Biases. Die Wahrscheinlichkeiten für die Ausgabe des Neuronalen Netzes werden durch $\widehat{y}_i$ bezeichnet.

$$L(W, b) = -\Sigma_{i=1}^{N} y_i \times log(\widehat{y}_i) + (1 - y_i) \times log(1 - \widehat{y}_i) \tag{4.5}$$

4.4 Evaluation

Um festzustellen, wie erfolgreich das Neuronale Netze bei einer Klassifikation ist, wird ein geeignetes Maß benötigt. Ein einfaches Mittel zur Evaluation von Klassifikationen ist die Konfusionsmatrix in Abbildung 4.3 [50]. Anhand dieser Tabelle ist es möglich die Ergebnisse einer binären Klassifikation richtig zu bezeichnen [50]. Ein „true positive“ bezeichnet ein korrekt erkanntes positives Ergebnis. Im Gegensatz dazu bezeichnet ein „false positive“ ein negativ erkanntes positives Ergebnis. Sollte ein negatives Ergebnis als ein positives erkannt werden, dann nennt man dies „false negative“. Der letzte Fall, ein „true negative“, tritt dann auf, wenn ein negatives Ergebnis auch korrekt als ein negatives Ergebnis erkannt wurde.

	P' (Predicted)	N' (Predicted)
P (Actual)	True Positive	False Negative
N (Actual)	False Positive	True Negative

Abbildung 4.3: Konfusionsmatrix aus [50]

Diese Matrix ist ideal für die Beschreibung einzelner Ergebnisse. Allerdings ist für Machine Learning ein Maß interessant, welches die korrekte Klassifikation aller Ergebnisse beschreibt. Ein Maß dafür ist die Genauigkeit („Accuracy“) [50]. Dieses Maß beschreibt, wie nah das gemessene Ergebnis an dem erwarteten Ergebnis liegt [50]. Gleichung (4.6) zeigt die entsprechende Formel [50].

$$Accuracy = (TP + TN)/(TP + FP + FN + TN) \tag{4.6}$$

Bei dieser Formel bezeichnet TP die Anzahl der „true positives“ und TN die Anzahl der „true negatives“. Dementsprechend beschreibt FN die Anzahl der „false negatives“ und FP die Anzahl der „false positives“. Allerdings gilt es zu berücksichtigen, dass die Genauigkeit irreführend sein kann, wenn die Anzahl der Beispiele für die Klassen zu stark voneinander abweicht [50]. So erhält man automatisch eine hohe Genauigkeit, da die kleinere Klasse zu selten vorkommt [50].

4.5 Embeddings

Wie aus Abschnitt 4.1 und 4.2 hervorgeht arbeiten Neuronale Netze mit Vektoren. Viele Informationen, wie Text, können daher nicht direkt an ein Neuronales Netz gegeben werden und müssen erst vektorisiert werden. Um Text in Neuronale Netzen verwenden zu können gibt es zwei Möglichkeiten.

Die erste Möglichkeit ist als „One Hot Encoding“ bekannt [25]. Dabei definiert die Größe des Wortschatzes die Größe des Vektors, da ein Wort einer Spalte im Vektor entspricht [25]. Sollte ein Wort in dem Text vorkommen, dann wird der entsprechende Wert im Vektor auf 1 gesetzt [25]. Sollte ein Wort nicht existieren, dann ist der Wert 0 [25]. Diese Vektoren können dann an ein Neuronales Netz gegeben werden. Allerdings ist diese Darstellung problematisch, da man dadurch sehr große Vektoren erhält und die natürliche Ordnung des Dokuments verloren geht [25].

Eine bessere Transformation ist die Abbildung von Daten auf einen Vektorraum, welcher als Embeddding bezeichnet wird [25]. Dabei handelt es sich um einen Vektorraum mit wenigen Dimensionen, welcher die Daten beschreibt [25]. Bei einem Word Embedding wird ein Wort durch einen Vektor in diesem Vektorraum beschrieben [37]. Idealerweise befinden sich ähnliche Datensätze an ähnlichen Positionen im Vektorraum, wodurch ein Neuronales Netz besser generalisiert [25]. Abbildung 4.4 macht den Vorteil dieser Repräsentaton deutlich. Aufgrund der Vektorisierung können Gemeinsamkeiten zwischen den Datensätzen abgeleitet werden [25]. Ein Neuronales Netz ist in der Lage diese Embeddings zu lernen oder bereits vorab trainierte Repräsentationen zu verwenden [25]. Dadurch kann das Ergebnis verbessert und der Trainingsaufwand reduziert werden [25].

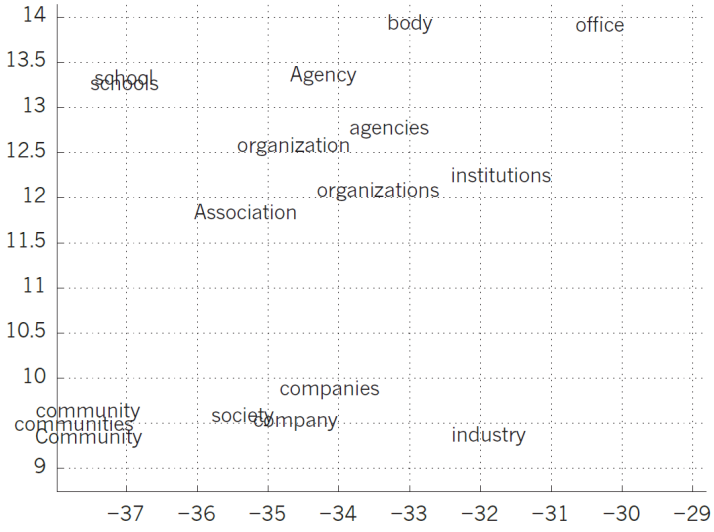

Abbildung 4.4: 2D Darstellung eines Word Embeddings aus [37]

4.6 Rekurrente Neuronale Netze

Traditionelle Neuronale Netze wie sie in Abbildung 4.2 zu sehen sind haben allerdings zwei Limitierungen. Die erste Limitierung ist, dass sie nur mit Eingabevektoren fester Länge arbeiten können [37]. Zweitens sind sie nicht in der Lage mit Daten zu arbeiten, welche das Konzept von Zeit verwenden, da ein Neuronales Netz immer nur die aktuelle Eingabe betrachtet [50]. Dies limitiert die Anwendungsmöglichkeiten von Neuronalen Netzen.

Eine andere Form von Feed-Forward Netzwerken sind Rekurrente Neuronale Netze (RNN) [50]. Rekurrente Neuronale Netze verwenden wiederkehrende Verbindungen, um Abhängigkeiten über Zeit zu modellieren [50]. Anstelle eines Eingabevektors mit fester Länge arbeiten RNNs mit mehreren Eingabevektoren unterschiedlicher Länge [50]. In einem RNN verfügen Neuronen der versteckten Ebenen über eine Ausgangsverbindung, welche gleichzeitig die Eingangsverbindung des gleichen Neurons ist [50]. Dieses Konzept ist in Abbildung 4.5 dargestellt. Dies erlaubt einem Neuron vorhergehende Eingaben zu berücksichtigen [50].

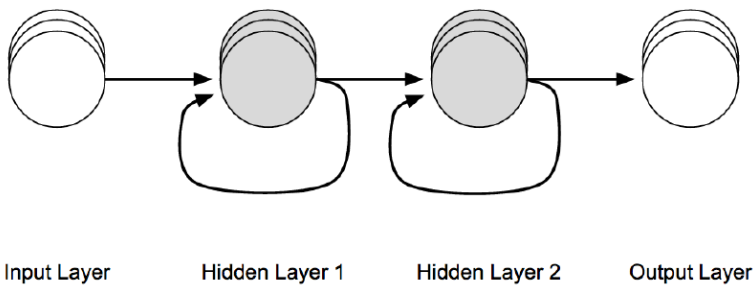

Abbildung 4.5: Rekurrente Verbindungen aus [50]

Durch die Implementierung der wiederkehrenden Verbindungen können Rekurrente Neuronale Netze gut mit zeitbasierten Daten arbeiten [50]. Audio oder Text sind Beispiele für Daten dieser Art [37] [50]. Die Daten in dieser Domäne sind geordnet und hängen von den vorhergehenden Daten ab [50].

Rekurrente Neuronale Netze werden durch ein Verfahren namens „Backpropagation through time“ trainiert [50]. Dabei handelt es sich im Grunde um Backpropagation, allerdings wird der Fehler für jeden Zeitschritt berechnet [50]. Um die partiellen Ableitungen für die Rekurrenten Neuronen zu berechnen kann die Kettenregel angewandt werden, da es sich bei einem Neuronalen Netz um eine Verkettung von Funktionen handelt [50]. Das kann allerdings dazu führen, dass der Gradient zu klein wird [50]. Dies wird auch als Vanishing Gradient bezeichnet, wodurch eine Anpassung der Gewichte kaum oder gar nicht mehr möglich ist [50].

4.7 Long-Short-Term Memory

Eine Möglichkeit, dass Problem des verschwindenden Gradienten zu überwinden, ist die Verwendung von Long-Short-Term Memory (LSTM) Einheiten [50]. Dabei handelt es sich um die meistgenutzte Variation von Rekurrenten Neuronalen Netzen [50]. LSTM Netzwerke wurden 1997 von Hochreiter und Schmidhuber eingeführt [31]. Eine LSTM Einheit verfügt über Verbindungen der vorhergehenden Ebene und des vorhergehenden Zeitschritts. Abbildung 4.6 zeigt den Aufbau der ursprünglichen LSTM Einheit, da mittlerweile verschiedene Variationen mit unterschiedlichen Implementierungen und Verbesserungen existieren.

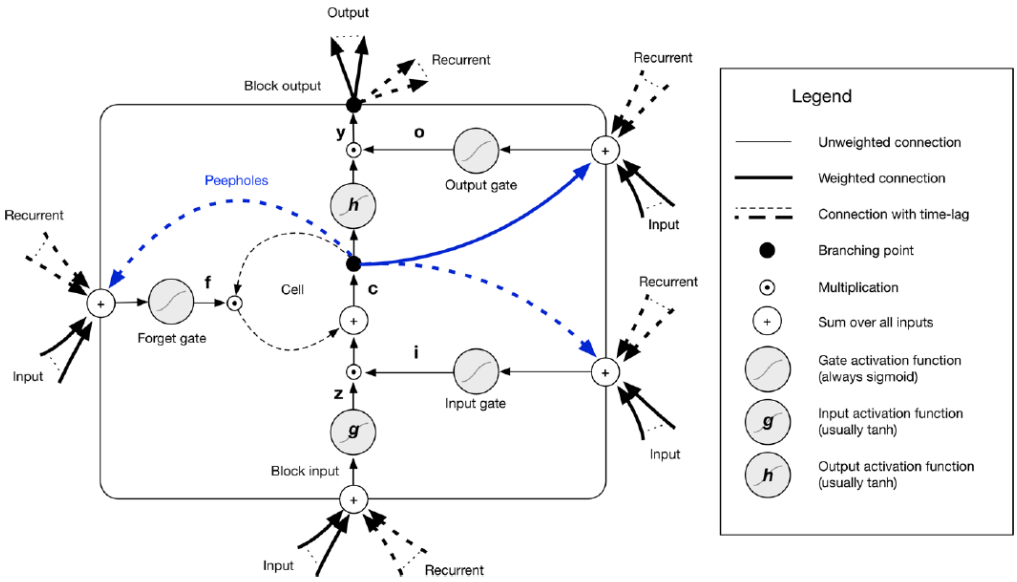

Abbildung 4.6: Abbildung einer LSTM Einheit aus [50]

LSTM Neuronen basierend auf einer einfachen Idee: ein Speicher, welcher über entsprechende Gates reguliert wird. Durch den Speicher kann der Zustand des Neurons über mehrere Zeitschritte beibehalten werden. Damit kann der Gradient über mehrere Zeitschritte moduliert werden, wodurch das Problem des verschwindenden Gradienten überwunden wird.

Insgesamt gibt es drei Gates: das Input Gate, das Forget Gate und das Output Gate. Das Input Gate reguliert, welche Eingaben für den Speicher relevant sind. Das Forget Gate hingegen reguliert, ob der Zustand für einen gegeben Zeitschritt beibehalten oder vergessen werden soll. Das Output Gate entscheidet, ob das Neuron aktiviert wird. Alle Gates verwenden dafür die aktuellen Eingaben des Netzwerks und die des vorhergehenden Zeitschritts für die Entscheidungen. In Gleichung (4.7) ist die mathematische Beschreibung für ein LSTM Neuron zu sehen.

$$
\begin{aligned}
\mathbf{z}^t &= g(\mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z) \\
\mathbf{i}^t &= \sigma(\mathbf{W}_i\mathbf{x}^t + \mathbf{R}_i\mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i) \\
\mathbf{f}^t &= \sigma(\mathbf{W}_f\mathbf{x}^t + \mathbf{R}_f\mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f) \\
\mathbf{c}^t &= \mathbf{i}^t \odot \mathbf{z}^t + \mathbf{f}^t \odot \mathbf{c}^{t-1} \\
\mathbf{o}^t &= \sigma(\mathbf{W}_o\mathbf{x}^t + \mathbf{R}_o\mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o) \\
\mathbf{y}^t &= \mathbf{o}^t \odot h(\mathbf{c}^t)
\end{aligned}
\tag{4.7}
$$

Dabei beschreibt $\mathbf{x}^t$ den Eingabevektor zu Zeitpunkt t, $\mathbf{W}$ die Gewichtmatrix der Eingabe, $\mathbf{R}$ die Gewichtmatrix der wiederkehrenden Eingaben und $\mathbf{b}$ den Bias Vektor. Die Funktionen σ, g und h bezeichnen Aktivierungsfunktionen. Dabei wird üblicherweise die Sigmoidfunktion für die Aktivierung der Gates verwendet und der Hyperbeltangens für die Aktivierung der LSTM Ein- und Ausgabe.

4.8 Forschungsstand

Die Extraktion semantischer Informationen von Quellcode ist bereits das Ziel mehrerer Arbeiten gewesen [3, 5, 6, 52]. Im Gegensatz zu dieser Arbeit geht es allerdings darum die semantischen Informationen vom Quellcode abzuleiten. Die vorliegende Arbeit kehrt diese Vorgehensweise um, indem zuerst semantische Eigenschaften formuliert werden und im Anschluss eine mögliche Extraktion geprüft wird. Soweit dem Autor bekannt ist dies die erste Arbeit, welche eine derartige Prüfung für Quellcode versucht.

Allerdings gibt es einige ähnliche Arbeiten, die sich mit der Idee der semantischen Extraktion zur Bewertung von Quellcode durch Machine Learning auseinander gesetzt haben. So konnte Bielik et al. 2016 zeigen, dass das Training eines statistischen Linters durch das Lernen von Analyseregeln möglich ist [12]. Sharma et al. konnte 2019 zeigen, das Code Smells durch Deep Learning identifiziert werden können, aber der Erfolg der Identifizierung vom Code Smell und der Deep Learning Architektur abhängt [56]. Einige Arbeiten haben sich mit dem Finden von Entwurfsmustern in Quellcode beschäftigt [2, 18, 22, 59], welche ebenfalls positive Ergebnisse aufzeigen können.

Abseits der Zielsetzung ist die ideale Repräsentation des Quellcodes in Neuronalen Netzen immer noch eine Herausforderung [5]. Da Hindle 2012 [29] zeigte, das Programmiersprachen ähnliche statistische Eigenschaften wie Text aufzeigen, konzentrierten sich viele Arbeiten auf die Anwendung von Deep Learning Ansätze für Text, um Informationen aus Quellcode zu gewinnen [4, 29, 53]. Dabei wurde Quellcode als Text behandelt und in entsprechende Token zerteilt, welche als Sequenz an ein Neuronales Netz weitergegeben wurden.

Durch diese Form der Repräsentation gehen allerdings wichtige semantische Informationen von Quellcode verloren. Die Semantik von Quellcode wird besonders durch dessen Struktur beschrieben, da beispielsweise die Reihenfolge von Methodenaufrufen nicht zufällig gewählt ist. Aus diesem Grund verwenden viele Arbeiten mittlerweile Abstrakt Syntax Trees zur Repräsentation von Quellcode [5, 47, 60]. Die Konstruktionen dieser Repräsentationen unterschiedet sich allerdings deutlich. Zhang et al. beispielsweise teilt einen Abstract Syntax Tree in kleinere Sequenzen auf, um Quellcode besser zu repräsentieren und die Probleme bei der Modellierung von längeren Abhängigkeiten zu minimieren [60]. Alon et al. hingegen repräsentieren einen Abstract Syntax Tree durch die einzelnen Pfade eines Trees [5, 6]. Dabei werden die Pfade in einem AST als Sequenzen modelliert und an das Neuronale Netze weitergegeben [6].

Auch wenn die Repräsentation durch ASTs zu guten Ergebnissen führt zeigte Tufano et al., dass unterschiedliche Repräsentationen zu unterschiedlichen Ergebnissen führen und das verschiedene Repräsentationen sich komplementieren [57]. Hinzu kommt, dass Ben-Hun et al. ebenfalls sehr gute Ergebnisse zeigen konnte, indem Quellcode in eine Zwischenrepräsentation abstrahiert wurde [10]. Dies macht deutlich, dass die Repräsentation von Quellcode zum Problem passen muss.

Die Wahl der Deep Learning Architektur ist ebenfalls abhängig von dem zu lösenden Problem. Während Sharma et al. mit Recurrent Neural Networks und Convolutional Neural Networks arbeiteten nutzen Alon et al. [6] und Xu et al. wiederum Neural Attention Networks [58]. Allamanis et al. [3] und Chen et al. [15] konnten ebenfalls gute Ergebnisse mit Graph Neural Networks und Tree-based Convolutional Neural Networks erreichen. Dies macht deutlich, dass unterschiedliche Architekturen für ähnliche Aufgaben verwendet werden können. Ähnlich wie bei der Repräsentation von Quellcode hat die Wahl der Architektur einen Einfluss auf das Ergebnis.

5 Methodik

Um festzustellen, ob semantische Merkmale von Quellcode durch Deep Learning extrahiert und für die Bewertung in einem Code Review oder Linter verwendet werden können, bedarf es einem geeignetem Untersuchungsdesign. Aus diesem Grund verwendet diese Arbeit einen quantitativen Forschungsansatz, um die Erfolgsrate einer Klassifikation von Quellcode zu messen. Zu Beginn werden daher einige Forschungsfragen in Abhängigkeit der Zielsetzung und auf Basis der Naturalness Theorie aus Abschnitt 2.3 formuliert. Im Anschluss wird beschrieben, wie sich die Stichprobe für die automatische Klassifikation zusammensetzt und halbautomatisch erhoben wurde. Zuletzt wird beschrieben, welche Maßnahmen zur Aufbereitung ergriffen werden, um die Daten in einem Neuronalen Netz zu verwenden, damit die daraus resultierenden Ergebnisse statistisch ausgewertet werden können.

5.1 Forschungsfragen

Der folgende Abschnitt dokumentiert die Forschungsfragen, welche auf Basis der Zielsetzung aus Kapitel 1 formuliert wurden. Außerdem wird die Motivation hinter den einzelnen Forschungsfragen beschrieben.

F1) Wie erfolgreich ist die Klassifikation von semantischen Eigenschaften einer Methode durch Deep Learning?

Wie bereits in Abschnitt 1.2 beschrieben wurde, ist der erste Schritt von Code Reviews und Lintern die Identifikation eines Problems. Ähnlich wie NLP Modelle könnten probabilistische Quellcode Modelle in der Lage sein, semantische Informationen zu extrahieren, welche mit binärer Klassifikation problematischen Quellcode ermitteln könnten. Daher prüft diese Forschungsfrage die Machbarkeit der binären Klassifikation durch das Training eines probabilistischen Deep Learning Modells. Dafür wird eine Stichprobe von Java Methoden auf Basis der Codequalität in Kapitel 3 angelegt, was dadurch begründet wird, dass eine objektorientierte Anwendung ihr Verhalten über Klassen und deren Operationen definiert. Tabelle 5.2 zeigt die für diese Untersuchung ausgewählten Merkmale für „gute“ Methoden. Eine Verletzung dieser Merkmale klassifiziert eine Methode als „schlecht“. Im Anschluss wird ein Neuronales Netz für Textklassifikation mit der Stichprobe trainiert und dessen Erfolg gemessen, um den Erfolg eines solchen Ansatzes zu messen.

Merkmal	Abschnitt
Die Methode ist möglichst klein	3.1.2
Die Methode hat nur eine Aufgabe	3.1.2
Die Methode enthält keinen doppelten Quellcode	3.1.2
Die Methode hat keine Flag Argumente	3.1.2
Die Methode berücksichtigt Command-Query-Separation	3.1.2
Die Methode verwendet keine Message Chains	3.2.5
Die Methode extrahiert oder verwendet kurze try/cache Blöcke	3.1.4
Die Methode verwendet kein Switch Case	3.2.2
Die Methode verwendet eine kurze Parameterliste	3.1.2
Die Methode ist nicht leer (Dead Code)	3.2.4
Die Methode gibt kein null zurück	3.1.4

Tabelle 5.1: Merkmale guter Methoden für Forschungsfrage 1.2

F2) Wie erfolgreich ist die Klassifikation von Methodennamen durch Deep Learning?

Ein weiteres wichtiges Merkmal von guten Methoden, welches durch diese Forschungsfrage separat betrachtet wird, ist die Verwendung sinnvoller Namen. Ähnlich wie bei der vorhergehenden Frage könnte hier die gleiche Begründung und das gleiche Vorgehen positive Ergebnisse zeigen, welches es zu überprüfen gilt. Für diese Forschungsfrage wird eine separate Stichprobe aus Methodennamen angelegt, welche gute und schlechte Methodennamen enthält. Diese Stichprobe wird anhand der Richtlinien für gute Namen in Abschnitt 3.1.1 zusammengestellt. Tabelle 5.2 zeigt eine zusammengefasste Liste der Merkmale guter Methodennamen. Auch bei dieser Forschungsfrage werden „schlechte“ Namen durch eine Verletzung dieser Richtlinien gekennzeichnet.

Merkmal	Abschnitt
Der Name enthält ein Verb oder eine Verbphrase	3.1.2
Der Name verwendet den korrekten Präfix, sofern angebracht	3.1.2
Der Name entspricht der Java Code Konvention	3.1.1
Der Name enthält keine Encodings	3.1.1
Der Name verwendet keine Abkürzungen	3.1.1

Tabelle 5.2: Kriterien guter Methodennamen für Forschungsfrage 1.1

F3) Welche statistischen Merkmale zeigen die Stichproben?

Da Deep Learning die relevanten Merkmale für die Klassifikation selbst extrahiert, ist keine manuelle Extraktion notwendig. Dennoch ist ein ausreichendes Verständnis der Trainingsdaten wichtig, um spätere Ergebnisse sinnvoll interpretieren zu können. Aus diesem Grund stellt sich die Frage nach statistischen Merkmalen der Trainingsdaten, welche als Grundlage für das Machine Learning Modell dienen. Diese Frage

soll mit Hilfe von Maßen der deskriptiven Statistik geeignete Daten liefern. Um einen tieferen Einblick in die Zusammensetzung der Stichprobe zu erhalten, wird die Häufigkeit der Token und ihr Durchschnitt ermittelt, um die Zusammenstellung der Stichprobe zu beurteilen.

5.2 Stichprobe

Als Grundlage für die Klassifikation wird eine ausreichend große Stichprobe benötigt. Für andere Deep Learning Probleme gibt es bereits standardisierte Stichproben wie den MNIST Datensatz [1]. Auch für Big Code gibt es einige Datensätze [2] [3]. Allerdings gibt es zum aktuellen Zeitpunkt keine standardisierten Stichproben, die den Anforderungen für Codequalität genügen, wie sie in Kapitel 3 beschrieben werden.

Aus diesem Grund wurde für diese Arbeit eine eigene Stichprobe zusammengestellt. Dadurch lässt sich nicht nur ein gewisses Maß an Qualität gewährleisten, sondern schafft auch ein besseres Verständnis der Stichprobe, was für die Interpretation der Deep Learning Ergebnisse vorteilhaft ist. Für die Stichprobe wird der Quellcode der Projekte aus Tabelle 5.3 verwendet. Diese Projekte wurden aufgrund ihrer Beliebtheit ausgewählt. Github erlaubt es Projekte zu favorisieren (sogenannte „Stars“), welche auf der Plattform als Indikator für Beliebtheit gelten.

Name	**Version**	**Lizenz**	**Beschreibung**
Apache Dubbo	2.7.3	Apache 2.0	RPC Framework
elasticsearch	7.4.2	Apache 2.0	RESTful Search Engine
EventBus	3.1.1	Apache 2.0	Event Bus für Android und Java
Gson	2.8.5	Apache 2.0	JSON De-/Serialisierung
Guava	28.1	Apache 2.0	Google Kernbibliothek für Java
okhttp	4.2.1	Apache 2.0	HTTP Client
RxJva	2.2.14	Apache 2.0	Reactive Extension für JVM

Tabelle 5.3: Tabellarische Aufstellung der Stichprobe

5.3 Datenerhebung

Um die Namen und Methoden zu extrahieren wurde sich für ein halbautomatisches Verfahren entschieden. Dadurch wird die Zusammenstellung des Datensatzes beschleunigt und gleichzeitig der Einfluss einer automatischen Extraktion auf den Datensatz minimiert. Da für eine automatische Extraktionen konkrete Muster definiert werden müssten, um diese zu extrahieren, würden sich so Muster im Datensatz

[1] MNIST Database of handwritten digits: `http://yann.lecun.com/exdb/mnist/`
[2] Learning from „Big Code“ Datasets: `http://learnbigcode.github.io/datasets/`
[3] Source{d} Datasets: `https://github.com/src-d/datasets`

widerspiegeln. Durch einen halbautomatischen Ansatz entscheidet immer noch ein Mensch, ob es sich um eine gute Methode (oder Namen) handelt oder nicht.

Zu diesem Zweck wurde ein eigenes Java Programm geschrieben, welches die relevanten Daten aus dem Quellcode extrahiert. Um dies zu erreichen, werden alle Java Dateien der Stichprobe ermittelt und deren Abstract Syntax Tree generiert. Dazu wird das Framework JavaParser[4] verwendet. Daraufhin werden die Methoden und deren Namen aus dem Abstract Syntax Tree entnommen, um diese über ein User Interface darzustellen. Abbildung 5.1 zeigt einen Screenshot des User Interfaces.

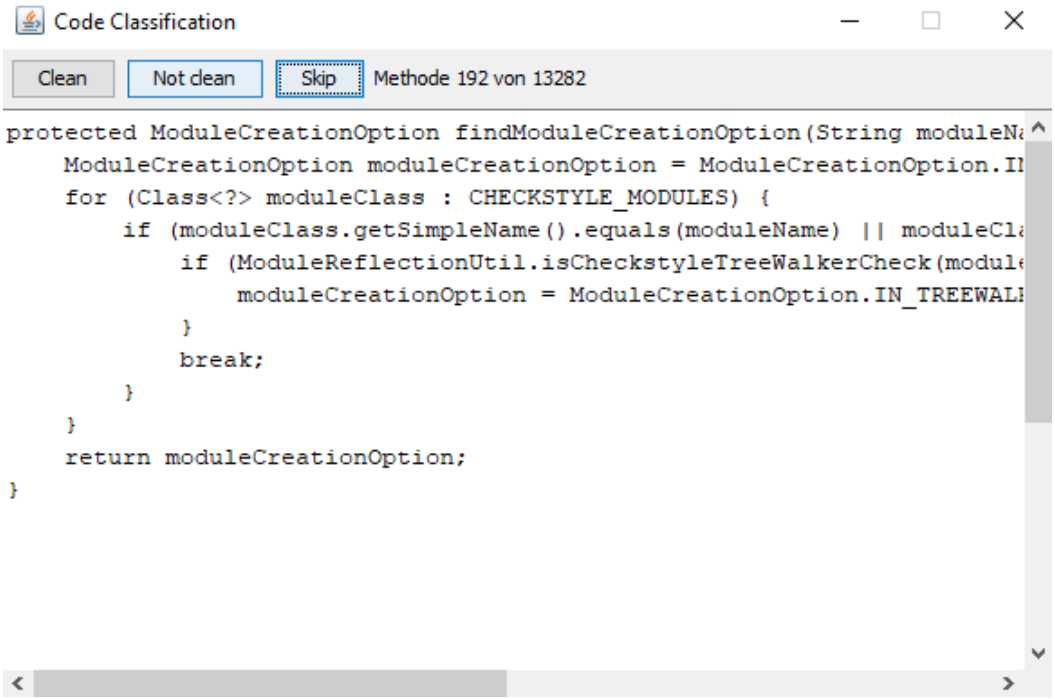

Abbildung 5.1: Das User Interface zur halbautomatischen Extraktion der Daten

Basierend darauf kann entschieden werden, ob die Methode oder der Name den definierten Richtlinien entspricht oder nicht. Die klassifizierten Merkmale werden im Anschluss automatisch in Textdateien abgelegt. In diesen Textdateien enthält jede Zeile ein einziges Merkmal. Durch die halbautomatische Analyse können die Methoden auch direkt für die Analyse durch ein Neuronales Netz vorbereitet werden. Zu diesem Zweck wurden einige Entscheidungen zur Normalisierung getroffen. So wurden Zeilenumbrüche und Tabs entfernt, damit die Methoden auf eine einzige Zeile passen. Zusätzlich wurden über Reguläre Ausdrücke vor und nach bestimmten Sonderzeichen (Klammern, Semikolon und Anführungszeichen) Leerzeichen eingefügt. Dadurch werden diese eindeutig von anderen Token getrennt und werden

[4] JavaParser: https://github.com/javaparser/javaparser

im Verlauf der Untersuchung als einzelne Token erfasst. Ein Beispiel dafür ist das Trennen von Klammern bei Parameterlisten einer Methode. Die Methode `public int getElement(int x)` würde sonst in die Token `public`, `int`, `getElement(int` und `x)` aufgeteilt werden. Durch die Vorverarbeitung werden nun die Token `public`, `int`, `getElement`, `(`, `int`, `x` und `)` erkannt. Außerdem können diese Daten durch die Formatierung nun von jedem beliebigen Programm eingelesen und Zeile für Zeile verarbeitet werden.

Die nun vorhandenen Token können jetzt als Eingabe für die Datenanalyse durch ein Neuronales Netz genutzt werden. Allerdings müssen diese vorher noch in Vektoren transformiert werden, damit ein Neuronales Netz in der Lage ist, diese zu verarbeiten. Aus diesem Grund werden die Token in eine Liste von Integern transformiert. Dafür wird jedes Token der Stichprobe eine eindeutige Zahl zugewiesen. Abbildung 5.1 zeigt ein Beispiel für eine Sequenz, welche eine Methode der Stichprobe kodiert.

```
# Tokensequenz
LoadingValueReference < K , V > insertLoadingValueReference ( ...

# Integersequenz
[212, 8, 16, 5, 15, 9, 2549, 2, ... ]
```

Listing 5.1: Codierung der Tokensequenz

Zuvor werden die Einträge der Stichprobe zufällig gemischt, um den Einfluss der halbautomatischen Erhebung weiter zu reduzieren. Im Anschluss wird die Stichprobe und in eine Trainings- und Validierungsprobe aufgeteilt. Als letztes werden die Sequenzen noch auf eine einheitliche Länge angepasst, damit Keras mit den Sequenzen arbeiten kann.

```
# split into training and test names
X_train = encoded_docs[0:int(len(encoded_docs)/2)]
y_train = labels[0:int(len(encoded_docs)/2)]

X_test = encoded_docs[int(len(encoded_docs)/2):]
y_test = labels[int(len(encoded_docs)/2):]

# truncate and pad input sequences
X_train = sequence.pad_sequences(X_train)
X_test = sequence.pad_sequences(X_test, maxlen=len(X_train[1]))
```

Listing 5.2: Aufteilen der Datensätze in Trainings- und Validierungsdatensätze

5.4 Datenanalyse

Das Neuronale Netz für die Analyse der Daten wird durch ein Framework namens Keras[5] umgesetzt, welches Tensorflow[6] als Backend nutzt. Tensorflow ist ein State of the Art Machine Learning Framework von Google. Keras ist ein Frontendframework, welches die Implementierung von Neuronalen Netzen erleichtert, indem es diverse andere Frameworks abstrahiert. Da Textklassifikation bereits eine gelöstes Problem für diese Frameworks darstellt, wird auf einem Tutorial von Tensorflow[7] aufgebaut. In Abbildung 5.3 ist der Quellcode zu sehen, welcher das Neuronale Netz in Python mittels Keras definiert. Dabei spricht man üblicherweise von einem *Model*, welches in Keras sequentiell definiert wird.

Das Neuronale Netz der Abbildung besteht insgesamt aus drei versteckten Ebenen, sowie einer Eingabe- und Ausgabeebene. Die „Embeddding"-Ebene stellt die erste versteckte Ebene des Netzes dar und erlaubt das Lernen des Embeddings. Die zweite versteckte Ebene besteht aus 64 LSTM Neuronen. Bei der dritten versteckten Ebene handelt es sich um eine Ebene mit 64 künstlichen Neuronen mit der ReLU Aktivierungsfunktion. Die Ausgabeebene ist ein einziges Neuron mit der Sigmoid Funktion als Aktivierungsfunktion für ein binäres Ergebnis.

```
# create the model
model = Sequential()
model.add(Embedding(number_of_token, 64, input_length=input_length)))
model.add(Bidirectional(LSTM(64)))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=[
    'accuracy'])

# print(model.summary())
history = model.fit(x_train, y_train, epochs=3)
```

Listing 5.3: Die Implementierung des Keras Models

Für das Training wird der ADAM Algorithmus verwendet [35]. Dabei handelt es sich um eine spezielle Form der gradientenbasierten Optimierung, welche rechnerisch effizient und einfach zu implementieren ist. Als Loss Funktion wird „binary crossentropy" verwendet. Zur Bestimmung des Erfolgs wird die Genauigkeit gemessen.

[5] Keras: `https://keras.io/`
[6] Tensorflow: `https://www.tensorflow.org/`
[7] Tensorflow - Text classification with an RNN: `https://www.tensorflow.org/tutorials/text/text_classification_rnn`

6 Ergebnisse

Die Ergebnisse zu den Forschungsfragen in Kapitel 5 sind insgesamt sehr unterschiedlich ausgefallen. Die nachfolgenden Abbildungen zeigen die komprimierten Ausgabe von Keras nach dem Training des Modells.

Forschungsfrage 1 dreht sich um den Gedanken, problematische Methoden durch Textklassifikation zu identifizieren. Nach dem Training des Neuronalen Netzes durch die aufgestellte Stichprobe konnte, ein Modell mit einer Präzision von **85.40** % trainiert werden. Dies ist ein recht gutes Ergebnis für Textklassifikation, allerdings muss man bedenken, dass die Stichprobe nur aus 2000 Beispielen besteht. Daher kann sich das Ergebnis verbessern oder verschlechtern, wenn man die gleiche Untersuchung mit einer größeren Stichprobe durchführt. Hinzu kommt, dass Quellcode dazu neigt, sich zu wiederholen, was einen positiven Effekt auf die Klassifikation haben kann.

```
 64/500 [..................] - loss: 0.6444 - accuracy: 0.8438
128/500 [==>...............] - loss: 0.6397 - accuracy: 0.8672
192/500 [=====>............] - loss: 0.6305 - accuracy: 0.8698
256/500 [========>.........] - loss: 0.6325 - accuracy: 0.8164
320/500 [==========>.......] - loss: 0.6241 - accuracy: 0.8062
384/500 [=============>....] - loss: 0.6137 - accuracy: 0.8125
448/500 [===============>..] - loss: 0.6070 - accuracy: 0.8170
500/500 [==================] - loss: 0.5935 - accuracy: 0.8220

Accuracy: 85.30%
```

Listing 6.1: Ergebnis des Keras Models für Methoden

Forschungsfrage 2 dreht sich um die binäre Klassifikation von Methodennamen. Diese Untersuchung wurde zweimal durchgeführt, da sich nach dem ersten Versuch für einen anderen Ansatz entschieden wurde. Bei dem ersten Versuch wurden die Namen anhand ihrer Buchstaben als Sequenz kodiert. So wurde jedem Buchstaben der Stichprobe eine einzigartige Nummer zugewiesen. Dies führte allerdings nur zu einem Ergebnis von **52.12** %. Dies ist ein schlechteres Ergebnis als bei Forschungsfrage 1, obwohl die Stichprobe mit 5000 Namen deutlich größer ist. Daher wurde sich für einen zweiten Versuch entschieden, bei dem die Namen nicht auf Buchstabenebene, sondern auf Tokenebene kodiert wurden. Das Ergebnis ist in Abbildung 6.3 zu sehen.

```
  64/2500 [..............] - loss: 0.6932 - accuracy: 0.4688
 128/2500 [>.............] - loss: 0.6913 - accuracy: 0.5469
 192/2500 [=>............] - loss: 0.6920 - accuracy: 0.5365
 256/2500 [==>...........] - loss: 0.6924 - accuracy: 0.5234
 ...
 704/2500 [====>.........] - loss: 0.6933 - accuracy: 0.4957
 768/2500 [=====>........] - loss: 0.6935 - accuracy: 0.4922
 832/2500 [=====>........] - loss: 0.6932 - accuracy: 0.4940
 896/2500 [======>.......] - loss: 0.6936 - accuracy: 0.4900
 960/2500 [=======>......] - loss: 0.6932 - accuracy: 0.4979
1024/2500 [========>.....] - loss: 0.6932 - accuracy: 0.5000
1088/2500 [=========>....] - loss: 0.6931 - accuracy: 0.5000
...
2368/2500 [============>..] - loss: 0.6925 - accuracy: 0.5000
2432/2500 [=============>.] - loss: 0.6925 - accuracy: 0.5021
2496/2500 [=============>.] - loss: 0.6925 - accuracy: 0.5016
2500/2500 [==============] - loss: 0.6925 - accuracy: 0.5012

Accuracy: 52.12%
```

Listing 6.2: Ergebnis des Namen Keras Models (Buchstaben)

Um die Namen als Token verwenden zu können mussten diese erst angepasst werden. Üblicherweise verwenden Java Methoden den „Camel Case“ als Konvention, weswegen einer typischer Name wie folgt aussieht: `getIdentifierName`. Aus diesem Grund wurden die Namen bei Auftreten eines Großbuchstaben aufgetrennt. Dies führte dazu, dass ein Methodenname durch eine Liste von Token beschrieben wurde, welcher im Anschluss mit dem gleichen Ansatz, wie bei Forschungsfrage 1, kodiert wurde. Allerdings verschlechterte sich das Ergebnis durch diese Änderung. Die Präzision liegt nun bei `47.20` %.

```
  64/2500 [..............] - loss: 0.6982 - accuracy: 0.4688
 320/2500 [==>...........] - loss: 0.6883 - accuracy: 0.5375
 512/2500 [=====>........] - loss: 0.6845 - accuracy: 0.5469
...
1088/2500 [======>.......] - loss: 0.6780 - accuracy: 0.5818
1280/2500 [=======>......] - loss: 0.6767 - accuracy: 0.5836
1472/2500 [========>.....] - loss: 0.6788 - accuracy: 0.5815
1664/2500 [=========>....] - loss: 0.6786 - accuracy: 0.5859
...
2112/2500 [==========>...] - loss: 0.6777 - accuracy: 0.5819
2304/2500 [===========>..] - loss: 0.6782 - accuracy: 0.5773
2496/2500 [=============>.] - loss: 0.6782 - accuracy: 0.5761
2500/2500 [==============] - loss: 0.6781 - accuracy: 0.5768

Accuracy: 47.20%
```

Listing 6.3: Ergebnis des Namen Keras Models (Token)

Die dritte und letzte Forschungsfrage dreht sich um die Zusammensetzung der Stichprobe. Zu diesem Zweck wurden die Token der Stichprobe gezählt und deren durchschnittliche Häufigkeit berechnet. In Abbildung 6.1 sind die Ergebnisse für die 15 häufigsten Token der Methoden zu finden. Eine größere Darstellung ist im Anhang 8.1 abgebildet. Hierbei lässt sich deutlich erkennen, dass ein Großteil der Token aus Sonderzeichen besteht. Erst ab Rang 10 folgen die ersten bezeichnenden Token, wie `return`. Sofern man die Top 100 Token untersucht, wird deutlich, dass übliche Java Token wie `int`, `String` oder `extends` zu Beginn der Tabelle auftreten. Erst danach folgen viele projektspezifische Token. Auch die durchschnittliche Häufigkeit fällt nach den ersten Token deutlich ab. So sind die Top 15 Token aus Abbildung 6.1 noch mit einer Mindesthäufigkeit von **1** % vertreten. Über die Top 50 Token hinaus machen die einzelnen Token nur noch weniger als **0,2** % des Datensatzes aus. Die Regelmäßigkeit und Häufigkeit der Sonderzeichen könnte ein Indiz für das gute Ergebnis von Forschungsfrage 1 sein, da viele projektspezifische Aufrufe nicht häufig genug auftauchen.

Nr.	**Token**	**Häufigkeit**	**Durchschnitt**
1	(	14913	10.889 %
2	)	14912	10.888 %
3	;	9232	6.741 %
4	,	5847	4.269 %
5	}	5830	4.257 %
6	{	5828	4.256 %
7	<	4088	2.985 %
8	>	4059	2.964 %
9	=	3431	2.505 %
10	return	2986	2.180 %
11	if	1861	1.359 %
12	public	1849	1.350 %
13	”	1783	1.302 %
14	V	1465	1.070 %
15	K	1405	1.026 %

Tabelle 6.1: Top 15 Token der Methoden aus Forschungsfrage 1

Zuletzt wird noch die Token der Methodennamen aus Forschungsfrage 2 untersucht. Eine Untersuchung der Buchstaben wird nicht durchgeführt, da eine Auflistung der Buchstaben als nicht aussagekräftig genug erachtet wird. In Abbildung 6.2 sind die Top 15 der Methodennamen zu finden. Im Anhang 8.2 befindet sich ebenfalls eine detailliertere Ansicht. In der Tabelle lässt sich auch direkt die Namenskonvention von Java Methoden erkennen, wie sie in Kapitel 3 beschrieben wurde. So sind die Token `get` und `set` auf den beiden ersten Plätzen und machen über **10** % eines Methodennamens aus.

Außerdem wird deutlich, dass Methoden, welche einen Zustand kommunizieren, den Präfix `is` erhalten. Die weiteren Ergebnisse der Abbildung heben auch hervor, dass häufig Verben für Methodennamen verwendet werden. Betrachtet man allerdings die Top 50 Token der Methodennamen, dann nehmen Nomen als Bezeichnung für Entitäten deutlich zu. Man erkennt zu Beginn, dass eine gewisse Menge an Verben sehr regelmäßig verwendet werden. Nach den Top 10 Token fällt die durchschnittliche Häufigkeit allerdings stark ab. Dies kann ein Indiz für das schlechte Ergebnis in Forschungsfrage 2 sein. Da Methodennamen aus sehr wenig Token besteht hat, das Neuronale Netz scheinbar nicht genug Informationen, um eine richtige Entscheidung zu treffen. Hinzu kommt, dass diese Informationen über zu wenig einzigartige Merkmale verfügen, da viele Methoden die gleichen Verben verwenden.

Interessant ist außerdem, dass die durchschnittliche Häufigkeit der beiden Stichproben recht unterschiedlich ausfällt. Obwohl die Stichprobe der Namen deutlich größer ist, enthält sie eine deutlich geringere Variation der Token. Die Stichprobe der Methoden hingegen hat eine größere Variation, obwohl sie über 3000 weniger Beispiele verfügt. Dies könnte ein Indiz dafür sein, dass das Neuronale Netz nicht die allgemeine Struktur von Quellcode gelernt hat, sondern nur die Struktur der Beispiele. Das könnte bedeuten, dass das Modell bei weiteren Daten schlecht generalisiert. Aus diesem Grund ist die Datenmenge nicht aussagekräftig genug, um eine definitive Aussage zu treffen.

Nr.	**Token**	**Häufigkeit**	**Durchschnitt**
1	get	1009	6.806 %
2	set	529	3.568 %
3	Null	447	3.015 %
4	On	341	2.300 %
5	is	217	1.464 %
6	Error	199	1.342 %
8	test	181	1.221 %
9	Value	148	0.998 %
10	create	140	0.944 %
11	do	135	0.911 %
12	add	123	0.830 %
13	Map	118	0.796 %
14	With	116	0.782 %
15	concat	114	0.769 %

Tabelle 6.2: Top 15 Token der Methodennamen aus Forschungsfrage 2

7 Fazit

Das Ziel dieser Arbeit war es festzustellen, ob es möglich ist, semantische Eigenschaften von Quellcode durch binäre Klassifikation zu erkennen. Wie sich nach der Zusammenstellung der Stichprobe zeigt, ist eine binäre Klassifikation mit einer Präzision von 85.40 % möglich. So kann ein gutes Ergebnis für Methoden mit unterschiedlichen Problemen erreicht werden. Sofern man über einen ausreichend großen Datensatz verfügt und eine kritische Menge an Problem erreicht, könnten Code Reviews oder Linter mit diesen Ergebnissen erweitert werden. Auch wenn dabei immer noch fast 20% der Fehler nicht erkannt werden, muss dies im Zusammenhang mit der Fehlererkennung von Lintern betrachtet werden. So sind Linter ebenfalls nicht in der Lage alle Probleme im Quellcode zu finden. Der Unterschied ist allerdings, dass ein Deep Learning Ansatz deutliche Verbesserungen und die Erschließung der semantischen Ebene verspricht. Im Gegensatz dazu, zeigte die Klassifikation von Methodennamen ein deutlich schlechteres Ergebnis mit einer Präzision von 52 %. Dies kann daran liegen, dass Methodennamen sich zu stark ähneln und deswegen nicht ausreichend voneinander abgegrenzt werden können.

7.1 Limitationen

Die Ergebnisse aus Kapitel 6 haben allerdings auch ihre Grenzen. Eine wichtige Limitierung liegt vor allem bei der Größe der Stichproben. Deep Learning ist sehr gut darin, die relevanten Features selbstständig zu extrahieren, allerdings muss es dafür die relevanten Features schon einmal gesehen haben. Aufgrund der kleinen Stichprobe dieser Arbeit können die Ergebnisse nicht für jede Form von objektorientierten Quellcode generalisiert werden. Deep Learning Projekte arbeiten in der Regel mit deutlich größeren Datenmengen. Um die hier präsentierten Ergebnisse zu generalisieren, wäre es daher notwendig einen weitaus größeren Datensatz anzulegen, welcher durchaus andere Ergebnisse produzieren könnte. Außerdem wurde hier nur eine binäre Klassifikation einer allgemeinen Motivation (gut vs. schlecht) untersucht. Ein Linter prüft hingegen spezifische Problemfälle für die ein Ergebnis mit Deep Learning schlechter ausfallen könnte.

Eine zweite Limitierung ist die Wahl der Architektur der Neuronalen Netze. Diese Arbeit basiert auf der Annahme, dass der Erfolg von Deep Learning mit Text sich auch auf Quellcode übertragen lassen könnte. Aus diesem Grund wurden aktuelle Architekturen für die Arbeit mit Text zur Klassifikation des Quellcodes verwendet. In den letzten Jahren ist aber im Bereich Deep Learning deutlich geworden,

dass sich mit unterschiedlichen Architekturen auch unterschiedliche Ergebnisse erreichen lassen. Die Menge an Architekturen ist allerdings so groß, dass diese nicht alle berücksichtigt werden können und die Wahl einer gängigen Architektur für Textklassifikation einen sinnvollen ersten Schritt darstellt.

Die letzte Limitierung der Ergebnisse ist die Wahl des Embeddings, welches einen Einfluss auf die Ergebnisse eines Neuronalen Netzes haben. Bisher gibt es keinen einheitlichen Ansatz für Embeddings von Quellcode. Bei der Arbeit mit Text konnte sich in den letzten Jahren Word2Vec hervorheben. Dabei handelt es sich um eine Gruppe von Modellen, mit denen sich Embeddings für Text generieren lassen. Um diese Modelle zu generieren, werden Neuronale Netze eingesetzt, um einen Vektorraum zu erzeugen, welcher die Wörter repräsentiert. Durch Word2Vec können einige Machine Learning Ansätze verbessert werden. Daher könnten die Ergebnisse dieser Arbeit mit einem anderen Embeddding besser ausfallen.

7.2 Ausblick

Das Forschungsfeld „Big Code“ ist noch neu und bietet daher viel Spielraum für weitere Ansätze und Forschungsergebnisse. Hinzu kommt, dass Deep Learning ein populäres Forschungsfeld ist und täglich neue Erkenntnis veröffentlicht werden. Aber wie es bei einem neuen Forschungsfeld zu erwarten ist, müssen viele Grundlagen in den nächsten Jahren geklärt werden.

Um überhaupt neue Ergebnisse zu erarbeiten ist die Zusammenstellung kurierter Stichproben notwendig. Mittlerweile existieren zwar einige relativ große Datensätze, allerdings sind diese auf allgemeine Untersuchungen ausgelegt. Da Deep Learning weitestgehend Supervised Learning Methoden verwendet, müssen Untersuchungen mit speziellen Anforderungen ihre eigenen Datensätze schaffen, welches die Erarbeitung neuer Ergebnisse behindert.

Außerdem müssen grundsätzliche Fragen, wie die Wahl der Embeddings und Architekturen, geklärt werden. In anderen Anwendungsgebieten ist der Forschungsstand mittlerweile soweit vorangeschritten, dass sich bestimmte Architekturen hervorheben konnten. Dies verdeutlicht das Deep Learning von einer einheitlichen Architektur profitieren würde, welche State of the Art Ergebnisse über verschiedene Anwendungsfelder hinweg erreicht.

Abseits der Grundlagenforschung bietet Big Code viele Anwendungsbereiche innerhalb der Softwareentwicklung. So können Anwendungsfelder wie Quellcodesynthese, -analyse, -übersetzung oder -optimierung von der Anwendung probabilistischer Modelle profitieren. All diese Anwendungsfelder können von den Erfolgen des NLP profitieren und versprechen einen Wandel in der Art und Weise wie Software in Zukunft entwickelt wird.

Literaturverzeichnis

[1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling - Vol.I: Parsing.* Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, 1972.

[2] Sultan Alhusain, Simon Coupland, Robert John, and Maria Kavanagh. Towards machine learning based design pattern recognition. In *2013 13th UK Workshop on Computational Intelligence (UKCI)*, pages 244–251. IEEE, 2013.

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. 2017.

[4] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code. 2016.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A General Path-Based Representation for Predicting Program Properties, 2018.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[7] Alberto Bacchelli and Christian Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.

[8] Vipin Balachandran. Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation. In *35th International Conference on Software Engineering (ICSE), 2013*, pages 931–940, Piscataway, NJ, 2013. IEEE.

[9] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern Code Reviews in Open-Source Projects. In *11th Working Conference on Mining Software Repositories : proceedings : May 31 - June 1, 2014, Hyderabad, India*, volume 11, 2014.

[10] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural Code Comprehension: A Learnable Representation of Code Semantics, 2018.

[11] Pavol Bielik, Veselin Raychev, and Martin Vechev, editors. *Programming with Big Code: Lessons, Techniques and Applications: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany*, 2015.

[12] Pavol Bielik, Veselin Raychev, and Martin Vechev. Learning a Static Analyzer from Data, 2016.

[13] Paul E. Black. Static Analyzers in Software Engineering. *The Journal of Defense Software Engineering*, 2009(3):16–17, 2009.

[14] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *A pattern language for distributed computing*, volume 4 of *Wiley series in software design patterns*. Wiley, Chichester, reprinted august 2011 edition, 2011.

[15] Long Chen, Wei Ye, and Shikun Zhang. Capturing source code semantics via tree-based convolution over API-enhanced AST. In Francesca Palumbo, Michela Becchi, Martin Schulz, and Kento Sato, editors, *Proceedings of the 16th ACM International Conference on Computing Frontiers - CF '19*, pages 174–182, New York, New York, USA, 2019. ACM Press.

[16] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon computing series. Yourdon Press, Englewood Cliffs, NJ, 21. [print] edition, 1979.

[17] Li Deng and Dong Yu. Deep learning: Methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[18] Ashish Kumar Dwivedi, Anand Tirkey, Ransingh Biswajit Ray, and Santanu Kumar Rath. Software design pattern recognition using machine learning techniques. In *2016 IEEE Region 10 Conference (TENCON)*, pages 222–227. IEEE, 2016.

[19] Karl Eilebrecht and Gernot Starke. *Patterns kompakt: Entwurfsmuster für effektive Softwareentwicklung*. IT kompakt. Springer Berlin Heidelberg, Berlin, Heidelberg, 5. aufl. 2019 edition, 2019.

[20] Pär Emanuelsson and Ulf Nilsson. A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 2008.

[21] Michael Fagan. Design and Code Inspections to Reduce Errors in Program Development. In Manfred Broy and Ernst Denert, editors, *Software Pioneers*, pages 575–607. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[22] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, and Janos Lele. Design pattern mining enhanced by machine learning. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 295–304. IEEE, 2005.

[23] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code // Improving the design of*

existing code. The Addison-Wesley object technology series. Addison-Wesley, Reading, MA, 1999.

[24] Erich Gamma. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, Boston, 39. printing edition, 2011.

[25] Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57(1):345–420, September 2016.

[26] Joachim Goll and Manfred Dausmann. *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java*. Springer Fachmedien Wiesbaden, Wiesbaden, 2013.

[27] Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. An overview on the Static Code Analysis approach in Software Development. 2009.

[28] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer Publishing Company, Incorporated, 2nd edition, 2012.

[29] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[30] Julia Hirschberg and Christopher D. Manning. Advances in natural language processing. *Science (New York, N.Y.)*, 349(6245):261–266, 2015.

[31] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[32] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92, 2004.

[33] S. C. Johnson. Lint, a C Program Checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[34] Kent Beck. *Test Driven Development*. Pearson Education (US), New Jersey, 2002.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014.

[36] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[37] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[38] P. M. Lewis and R. E. Stearns. Syntax-Directed Transduction. *Journal of the ACM*, 15(3):465–488, 1968.

[39] Barbara Liskov. Keynote address - data abstraction and hierarchy. In Leigh Power and Zvi Weiss, editors, *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum) - OOPSLA '87*, pages 17–34, New York, New York, USA, 1987. ACM Press.

[40] P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.

[41] M. V. Mantyla and C. Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[42] Robert C. Martin. *Agile software development, principles, patterns, and practices*. Pearson, Harlow, 1. ed., pearson new international ed. edition, 2014.

[43] Robert C. Martin, Michael C. Feathers, Timothy R. Ottinger, Jeffrey J. Langr, Brett L. Schuchert, James W. Grenning, and Kevin Dean Wampler. *Clean code: A handbook of agile software craftsmanship*. Robert C. Martin series. Prentice Hall, Upper Saddle River, NJ, 2009.

[44] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[45] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[46] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall international series in computer science. Prentice Hall, New York, [nachdr.] edition, 1993.

[47] Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, and Lu Zhang. Building Program Vector Representations for Deep Learning.

[48] Prakash M. Nadkarni, Lucila Ohno-Machado, and Wendy W. Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association : JAMIA*, 18(5):544–551, 2011.

[49] Meilir Page-Jones. *The practical guide to structured systems design*. Yourdon Press computing series. Yourdon Pr, Englewood Cliffs, N.J., 2. ed. edition, 1988.

[50] Josh Patterson and Adam Gibson. *Deep learning: A practitioner's approach*. O'Reilly, Sebastopol CA, first edition edition, 2017.

[51] Herbert Prahofer, Florian Angerer, Rudolf Ramler, Hermann Lacheiner, and Friedrich Grillenberger. Opportunities and challenges of static code analysis of IEC 61131-3 programs. In *Proceedings of 2012 IEEE 17th International*

Conference on Emerging Technologies & Factory Automation (ETFA 2012), pages 1–8. IEEE, 2012.

[52] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting Program Properties from Big Code. In Sriram Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*, pages 111–124, New York, New York, USA, 2015. ACM Press.

[53] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In Michael O'Boyle and Keshav Pingali, editors, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*, pages 419–428, New York, New York, USA, 2013. ACM Press.

[54] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957.

[55] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[56] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. On the Feasibility of Transfer-learning Code Smells using Deep Learning, 2019.

[57] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning Similarities from Different Representations of Source Code. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, pages 542–553, New York, New York, USA, 2018. ACM Press.

[58] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. Method name suggestion with hierarchical attention networks. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation - PEPM 2019*, pages 10–21, New York, USA, 2019. ACM Press.

[59] Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. On applying machine learning techniques for design pattern detection. *Journal of Systems and Software*, 103:102–117, 2015.

[60] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

8 Anhang

8.1 Top 100 Ergebnisse Forschungsfrage 3: Methoden

Nr	Token	Häufigkeit	Durchschnitt
1	(	14913	10.889 %
2	)	14912	10.888 %
3	;	9232	6.741 %
4	,	5847	4.269 %
5	}	5830	4.257 %
6	{	5828	4.256 %
7	<	4088	2.985 %
9	>	4059	2.964 %
10	=	3431	2.505 %
11	return	2986	2.180 %
12	if	1861	1.359 %
13	public	1849	1.350 %
14	”	1783	1.302 %
15	V	1465	1.070 %
16	K	1405	1.026 %
17	new	1342	0.980 %
18	?	1092	0.797 %
19	int	1091	0.797 %
20	null	1048	0.765 %
21	T	975	0.712 %
22	==	944	0.689 %
23	E	881	0.643 %
24	0	642	0.469 %
25	value	625	0.456 %
26	static	614	0.448 %
27	throw	562	0.410 %
28	+	555	0.405 %
29	String	550	0.402 %
30	else	515	0.376 %
31	boolean	511	0.373 %
32	@Override	498	0.364 %

33	e	496	0.362 %
34	key	487	0.356 %
35	!=	476	0.348 %
36	Object	465	0.340 %
37	i	412	0.301 %
38	extends	410	0.299 %
39	void	394	0.288 %
40	:	353	0.258 %
41	@Nullable	341	0.249 %
42	final	325	0.237 %
43	1	312	0.228 %
44	-	309	0.226 %
45	for	302	0.221 %
46	Entry	302	0.221 %
47	throws	301	0.220 %
48	checkNotNull	279	0.204 %
49	c	273	0.199 %
50	&&	257	0.188 %
51	case	249	0.182 %
52	IOException	248	0.181 %
53	index	246	0.180 %
54	true	242	0.177 %
55	Class	229	0.167 %
56	result	223	0.163 %
57	private	221	0.161 %
58	Iterator	217	0.158 %
59	instanceof	214	0.156 %
60	size	213	0.156 %
61	super	207	0.151 %
62	false	206	0.150 %
63	long	203	0.148 %
64	——	198	0.145 %
65	type	189	0.138 %
66	element	188	0.137 %
67	errorMessageTemplate	186	0.136 %
68	ReferenceEntry	183	0.134 %
69	entry	181	0.132 %
70	this	180	0.131 %
71	hash	179	0.131 %
72	catch	176	0.129 %
73	p1	176	0.129 %

74	try	175	0.128 %
75	C	172	0.126 %
76	while	159	0.116 %
77	Collection	157	0.115 %
78	p	155	0.113 %
79	p2	144	0.105 %
80	char	141	0.103 %
81	next	140	0.102 %
82	count	131	0.096 %
83	TypeAdapter	114	0.083 %
84	a	112	0.082 %
85	first	111	0.081 %
86	in	108	0.079 %
87	iterator	108	0.079 %
88	obj	105	0.077 %
89	Type	104	0.076 %
90	b	101	0.074 %
91	name	100	0.073 %
92	UnsupportedOperationException	100	0.073 %
93	lenientFormat	100	0.073 %
94	Node	97	0.071 %
95	of	96	0.070 %
96	JsonWriter	96	0.070 %
97	.append	95	0.069 %
98	List	94	0.069 %
99	t	94	0.069 %
100	i++	93	0.068 %

8.2 Top 100 Ergebnisse Forschungsfrage 3: Namen

Nr	Token	Häufigkeit	Durchschnitt
1	get	1009	6.806 %
2	set	529	3.568 %
3	Null	447	3.015 %
4	On	341	2.300 %
5	is	217	1.464 %
6	Error	199	1.342 %
7	test	181	1.221 %
8	Value	148	0.998 %
9	create	140	0.944 %
10	do	135	0.911 %
11	add	123	0.830 %
12	Map	118	0.796 %
13	With	116	0.782 %
14	concat	114	0.769 %
15	Index	111	0.749 %
16	For	101	0.681 %
17	Iterable	99	0.668 %
18	Next	96	0.648 %
19	To	92	0.621 %
20	remove	91	0.614 %
21	As	90	0.607 %
22	merge	88	0.594 %
23	From	82	0.553 %
24	Request	81	0.546 %
25	Type	81	0.546 %
26	assert	79	0.533 %
27	Scheduler	79	0.533 %
28	In	77	0.519 %
29	Function	74	0.499 %
30	Exception	70	0.472 %
31	subscribe	68	0.459 %
32	Throws	68	0.459 %
33	check	67	0.452 %
34	Delay	65	0.438 %
35	If	65	0.438 %
36	Returns	65	0.438 %
37	Single	59	0.398 %
38	Key	57	0.384 %
39	Set	56	0.378 %

40	All	55	0.371 %
41	Observable	54	0.364 %
42	And	54	0.364 %
43	Subscribe	53	0.358 %
44	Shard	53	0.358 %
45	Array	52	0.351 %
46	Entry	50	0.337 %
47	Field	50	0.337 %
48	One	49	0.331 %
49	Time	49	0.331 %
50	Queue	48	0.324 %
51	has	48	0.324 %
52	Last	47	0.317 %
53	Not	46	0.310 %
54	Shards	46	0.310 %
55	First	45	0.304 %
56	Normal	45	0.304 %
57	Complete	44	0.297 %
58	Empty	44	0.297 %
59	read	44	0.297 %
60	Of	43	0.290 %
61	By	43	0.290 %
62	After	42	0.283 %
63	Handler	42	0.283 %
64	Flowable	41	0.277 %
65	try	41	0.277 %
66	start	41	0.277 %
67	Settings	41	0.277 %
68	State	40	0.270 %
69	Name	38	0.256 %
70	Size	37	0.250 %
71	parse	37	0.250 %
72	Node	37	0.250 %
73	Response	37	0.250 %
74	Source	36	0.243 %
75	Stats	35	0.236 %
76	retry	34	0.229 %
77	skip	34	0.229 %
78	Count	34	0.229 %
79	Is	34	0.229 %
80	Dispose	33	0.223 %

81	flat	33	0.223 %
82	Reference	33	0.223 %
83	Assembly	32	0.216 %
84	Action	32	0.216 %
85	Timed	32	0.216 %
86	delay	31	0.209 %
87	drain	31	0.209 %
88	write	31	0.209 %
89	Iterator	31	0.209 %
90	using	30	0.202 %
91	clear	30	0.202 %
92	Failure	30	0.202 %
93	Task	30	0.202 %
94	take	29	0.196 %
95	Write	29	0.196 %
96	wait	29	0.196 %
97	dispose	28	0.189 %
98	Timeout	28	0.189 %
99	Or	28	0.189 %
100	Testing	28	0.189 %

Glossar

Abstract Syntax Tree Eine Baumstruktur, die den Quellcode repräsentiert und weniger bedeutende Strukturen (Klammern, Semikolons, ..) abstrahiert sowie die Iteration des Quellcodes erlaubt.

Aktivierung Bei einer Aktivierung handelt es sich um einen Ausgabewerte eines Neurons in einem Neuronalen Netz.

Aktivierungsfunktion Bei der Aktivierungsfunktion handelt es sich um eine mathematische Funktion, welche entscheidet, welcher Wert das Neuron innerhalb eines Neuronalen Netzes abgibt.

Backpropagation Bei Backpropagation handelt es sich um einen pragmatischen Ansatz, bei dem der Fehler eines Netzwerks auf die einzelnen Gewichte des Neuronalen Netzwerks verteilt wird.

Big Code Big Code bezeichnet die Anwendung von Big Data Methoden auf Quellcode.

Bottom-up Bottom-up beschreibt die Reihenfolge der Konstruktion eines Baumes bei der ein Baum von den Kindknoten zur Wurzel hin konstruiert wird.

Clean Code Bei Clean Code handelt es sich um eine Sammlung von Richtlinien, die guten objektorientierten Quellcode beschreiben.

Code Smells Code Smells sind wiederkehrende Fehlermuster, welche auf tiefer liegende Probleme des Quellcodes hinweisen.

Code Inspektion Bei Code Inspektionen handelt es sich um einen durch Fagan formalisierten Prozess, bei dem Quellcode durch Gruppenmeetings diskutiert und Zeile für Zeile anhand fester Kriterien analysiert werden, um Fehler zu finden.

Code Review Bei einem Code Review handelt es sich um eine anerkannte und informelle Methode zum Finden von Fehlern. Dabei wird der Quellcode regelmäßig durch andere Entwickler überprüft, um Fehler zu finden oder Standards durchzusetzen.

Deep Learning Ist ein Teilfeld von Machine Learning, welche sich mit Neuronalen Netzen beschäftigt, um Probleme auf Basis von Inferenz und ohne Handlungswissen zu lösen.

Design by Contract Bei „Design by Contract“ handelt es sich um einen Ansatz der Softwareentwicklung, bei der das Zusammenspiel einzelner Module durch die Formulierung von Vor- und Nachbedingungen gewährleistet wird. Zusätzlich existieren Grundannahmen die für eine Module über die ganze Objektlaufzeit hinweg gelten.

Dynamische Analyse Die dynamische Analyse prüft ein Programm auf Fehler, indem es dieses ausführt und dessen Verhalten überprüft.

Embeddding Ein Embedding ist eine Abbildung von diskreten Daten auf einen Vektor mit wenig Dimensionen.

Epochen Die Anzahl der Epochen beschreibt die maximale Anzahl der Iterationen eines Trainingsprozesses eines Neuronalen Netzes.

Feature Als Features bezeichnet man für Machine Learning relevante Merkmale, um die gegebene Aufgabe durch Inferenz zu bewältigen.

Learning Rate Die Learning Rate definiert die Schrittgröße für Anpassungen der Gewichte innerhalb eines Neuronalen Netzes.

Linter Ein Linter ist ein Programm, das Quellcode mittels Statischer Analyse auf Fehler überprüft.

Long-Short-Term Memory Ein Long-Short-Term-Memory Neuron ist ein spezielles Neuron von Rekurrenten Neuronalen Netzen, welches das Problem des verschwindenden Gradienten durch die Verwendung von Gattern löst.

Multilayer-Feed-Forward Netz Ein Multilayer-Feed-Forward Netz ist ein gerichteter Graph von Neuronen, bei dem die Ausgaben der Neuronen einer Schicht zur Eingabe der nächsten Schicht werden.

Natural Language Processing Natural Language Processing ist eine Querschnittsdisziplin, welche sich aus den Forschungsbereichen Computerlinguistik und künstlicher Intelligenz zusammensetzt und beschäftigt sich mit der Analyse von natürlichen Sprachen.

Neuron Ein Neuron ist die kleinste Einheit eines Neuronalen Netzes und wird durch eine mathematische Funktion modelliert.

Neuronales Netz Ein Neuronales Netz ist ein mathematisches Modell für Machine Learning, welches grob die Signalverarbeitung des menschlichen Gehirns zu modelliert.

Post-order Die Post-order ist eine Form der Traversierung eines Baumes. Dabei wird der Baum von der Wurzel hin zu den Kindknoten durchlaufen.

Pre-order Die Pre-order ist eine Form der Traversierung eines Baumes. Dabei werden erst die Knoten eines Baumes besucht und dann die Wurzel.

Rekurrente Neuronale Netze Rekurrente Neuronale Netze sind eine spezeille Form von Multilayer-Feed-Forward Netzen bei denen die einzelnen Neuronen über zirkuläre Verbindungen verfügen. Dadurch sind vorherige Eingabe zu späteren Zeitpunkten noch im Neuron präsent.

Representation Learning Als Representation Learning bezeichnet man die Fähigkeit eines Neuronalen Netzes die relevanten Merkmale von Daten selbstständig zu extrahieren.

Statische Analyse Die statische Analyse überprüft ein Programm auf Fehler, indem es den Quellcode analysiert. Dabei wird das Programm nicht ausgeführt, sondern Fehler durch die Erkennung von Mustern identifiziert.

Supervised Learning Das Training eines Neuronalen Netzes auf Basis vorher klassifizierter Datensätze bezeichnet man als Supervised Learning.

Token Token sind einzelne Bestandteile einer Zeichenkette, welche üblicherweise durch Leerzeichen getrennt werden.

Top-down Top-down beschreibt die Reihenfolge der Konstruktion eines Baumes bei der ein Baum von der Wurzel hin zu den Kindknoten konstruiert wird.

Unsupervised Learning Das Training eines Neuronalen Netzes auf Basis von Rohdaten bezeichnet man als Unsupervised Learning.